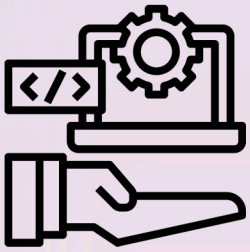
TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN

DISEÑO Y ADMINISTRACIÓN DE BASE DE DATOS



AÑO 2025

AUTOR: ALEJANDRO MOSTOVOI
PROFESOR: CRISTIAN BENITEZ

UTN

TODOS LOS DERECHOS RESERVADOS

Diseño y Administración de Base de Datos

Unidad I

Introducción a las Bases de Datos

Ing. Alejandro Mostovoi

Índice

Definición de un sistema de base de datos **2**
- Gestor de almacenamiento 3
- Procesador de consultas 3

Ventajas del enfoque de Bases de Datos **4**

Independencia de los Datos **5**

Historia y evolución de los sistemas de bases de datos. **7**
- Sistema de archivos (File Systems) 7
- Jerárquico (Hierarchical) 7
- Red (Network) 8
- Relacional (Relational) 9
- Objeto (Object) 11
- Objeto-Relacional (Object-Relational) 12
- NoSQL 12

Tipos de base de datos **12**
- Transaccionales 12
- Sistemas para la toma de decisión 13
- Híbridos 13
- OLTP vs OLAP 13
- Conclusión 14

Bibliografía **15**

Definición de un sistema de base de datos

I THINK WE SHOULD BUILD AN SQL DATABASE.
UH-OH
DOES HE UNDERSTAND WHAT HE SAID OR IS IT SOMETHING HE SAW IN A TRADE MAGAZINE AD?
WHAT COLOR DO YOU WANT THAT DATABASE?
I THINK MAUVE HAS THE MOST RAM.
© 1998 United Feature Syndicate, Inc.(NYC)

Definición	Un sistema **de base de datos**, también conocido como DBMS (DataBase Management System), es un conjunto de datos estructurados e interrelacionados junto con el software que permite acceder a dichos datos.

El objetivo principal de un sistema de base de datos es permitir almacenar y recuperar los datos así como también la manipulación de los mismos para producir información.

Podemos decir que una base de datos es un objeto estructurado que consta de 2 partes, por un lado una colección de datos relacionados y organizados en tablas (finalmente terminan siendo archivos físicos dentro una pc) y por otro una serie de metadatos que describen la estructura de los datos dentro de la base de datos. Por ejemplo, si quisiéramos almacenar datos de alumnos de un establecimiento educativo, el nombre, la edad y curso al que asiste serían los datos, mientras que el nombre que le asigne a cada atributo, el tipo de dato, longitud, etc. serían los metadatos.

Definición	Un modelo **de base de datos** es más un concepto que un objeto físico. Especifica cómo van a estar organizados y cómo se van a relacionar los datos dentro de la base de datos.

Definición: Componentes de un sistema de base de datos

Un sistema de base de datos está formado principalmente por 2 grandes componentes, por un lado el *gestor de almacenamiento* y por otro el *procesador de consultas.*

Gestor de almacenamiento

El gestor de almacenamiento es un módulo cuya función principal es almacenar, recuperar y actualizar los datos en la base de datos. Proporciona de forma indirecta una interfaz entre los datos (bajo nivel) y los programas de aplicación.

Este componente recibe instrucciones en DML (Data Manipulation Language) y las traduce a instrucciones de un sistema de archivos.

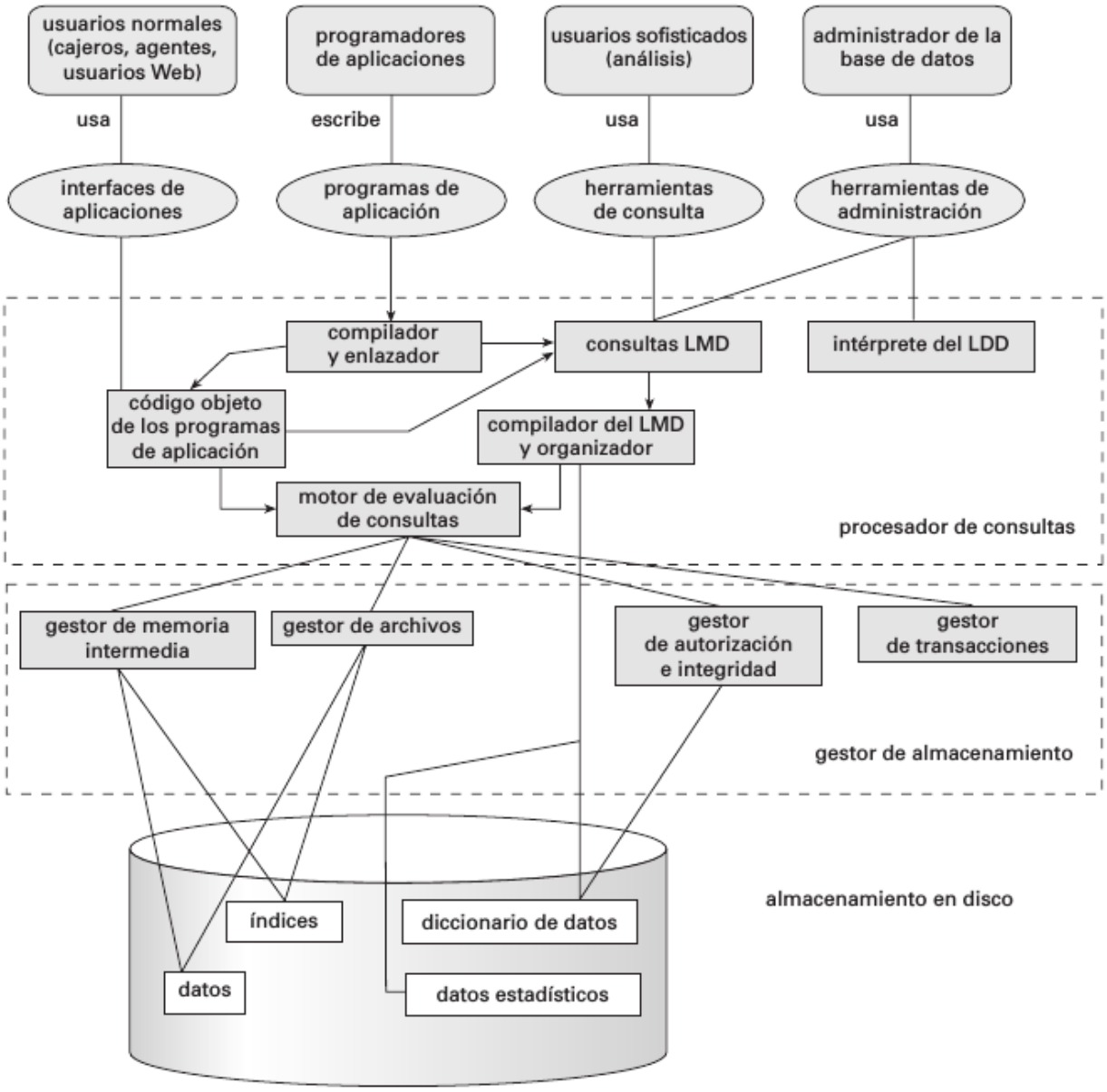
El gestor de almacenamiento, a su vez está compuesto por:

- Gestor de archivos, encargado de gestionar las operaciones contra el disco rígido
- Gestor de memoria, encargado de traer los datos desde el disco a la memoria principal (RAM). También gestiona el almacenamiento de datos en memoria caché

Procesador de consultas

Está conformado por:

- Intérprete de DDL (Data Definition Language)
- Compilador de DML, encargado de traducir instrucciones en DML a un plan de evaluación que se pasa al *motor de evaluación de consultas*. Puede generar más de 1 plan de evaluación de consultas y también puede producir algunas optimizaciones sobre las consultas.
- Motor de evaluación de consultas, encargado de ejecutar las instrucciones que recibe del *Compilador de DML*.



Ventajas del enfoque de Bases de Datos

Podemos decir que un sistema de base de datos ofrece, frente a un sistema de archivos, un control centralizado de los datos. A partir de esto se desprenden otras ventajas que se explican a continuación:

- **Compartir datos**

Una base de datos permite que varias aplicaciones puedan acceder a los datos y que se desarrollen nuevas aplicaciones que también pueden acceder a los datos sin tener que modificar la información contenida en la base de datos.

- **Reducir la redundancia**

  Sin una base de datos, cada aplicación posee sus propios archivos. Esto facilita la redundancia de los datos con el consecuente desperdicio de espacio de almacenamiento. No toda redundancia de datos es mala, hay casos en los que es necesaria pero debe estar controlada.

- **Evitar inconsistencias**

  Las inconsistencias entre los datos se dan principalmente cuando el mismo dato se encuentra representado en más de 1 entidad dentro de la base de datos. De esta manera en una entidad podría tener unos datos y en la otra entidad otros distintos. Las inconsistencias quedan prácticamente eliminadas si se aplica un control efectivo de la redundancia (o se elimina).

- **Manejo de transacciones**

  Una transacción es una unidad de trabajo lógica, que por lo general comprende varias operaciones, sobre la base de datos, permitiendo aplicar todas o ninguna de ellas.

- **Mantener la integridad de los datos**

  La integridad de los datos trata principalmente de la correctitud de los mismos dentro de la base de datos. La falta de integridad puede deberse a problemas de inconsistencia (producto de la redundancia), o bien por falta de controles de integridad, también conocidas como reglas de negocio.

- **Controlar la seguridad**

  Una base de datos permite asegurar que el único medio de acceso sea a través de los canales adecuados y definir restricciones de acceso que se verificarán en cada operación

Independencia de los Datos

La independencia de los datos es una ventaja o mejor dicho un objetivo buscado por los sistemas de bases de datos.

Definición	Podemos definir la independencia de los datos como la inmunidad de las aplicaciones a cambios en la representación física y técnicas de acceso de los datos.

Esto significa que dentro de la base de datos podemos cambiar la forma como se almacenan físicamente los datos, el tipo de indexado de los mismos, inclusive el dato indexado, etc. sin tener que modificar el código de la aplicación que accede a los mismos.

Ejemplo:

Imagine una aplicación que administre datos de alumnos de un establecimiento educativo. Si por motivos de rendimiento, se necesitará implementar un indexado de los datos sobre el campo *nombre*, la aplicación debería ser modificada para tal fin.

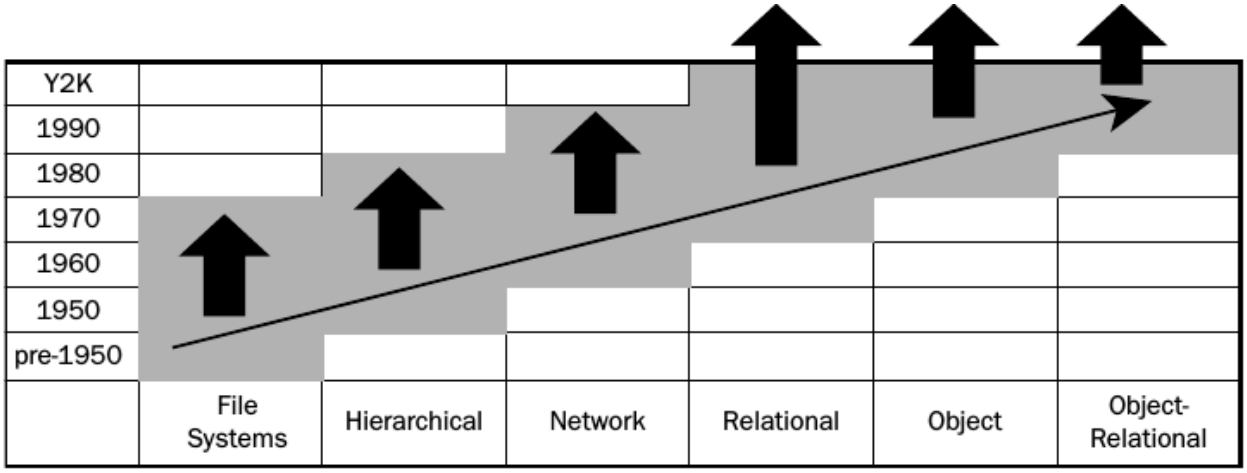
En un sistema de bases de datos, este cambio sería transparente para la aplicación, ya que el mismo se implementaría directamente en la base de datos sin modificar la aplicación.

Otros casos de independencia de datos podrían ser:

- **Representación de los datos**
  - Campos numéricos: podría ser necesario cambiar la base (binaria, decimal, etc); almacenar por un lado la parte decimal y por otro la parte entera, etc.
  - Campos alfanuméricos: cambiar la codificación de los caracteres (Unicode, UTF-8, ISO-8859-1, etc)
- **Codificación de los datos**
  - En algunos casos es deseable codificar los datos, por ejemplo un campo que guarda una cadena de caracteres Rojo, Verde, Azul podría codificarse como 1, 2, 3 respectivamente.
- **Materialización de los datos**
  - En ocasiones el dato que “ve” la aplicación no es necesariamente el que está almacenado en la base de datos. Ejemplos de este tipo surgen cuando el campo solicitado por la aplicación es producto de un cálculo que realiza la base de datos.

La independencia de los datos también implica que la base de datos pueda crecer sin tener que modificar las aplicaciones que acceden a la misma. Imagine que sobre una entidad *alumno* se agregan atributos como ser *telefono_celular, email*. Si estos atributos no son requeridos por una aplicación, la misma no debería verse afectada.

Historia y evolución de los sistemas de bases de datos.



Sistema de archivos (File Systems)

En este tipo de sistema, los datos son almacenados en archivos planos (texto o binarios) utilizando la estructura del sistema de archivos proporcionada por el sistema operativo.

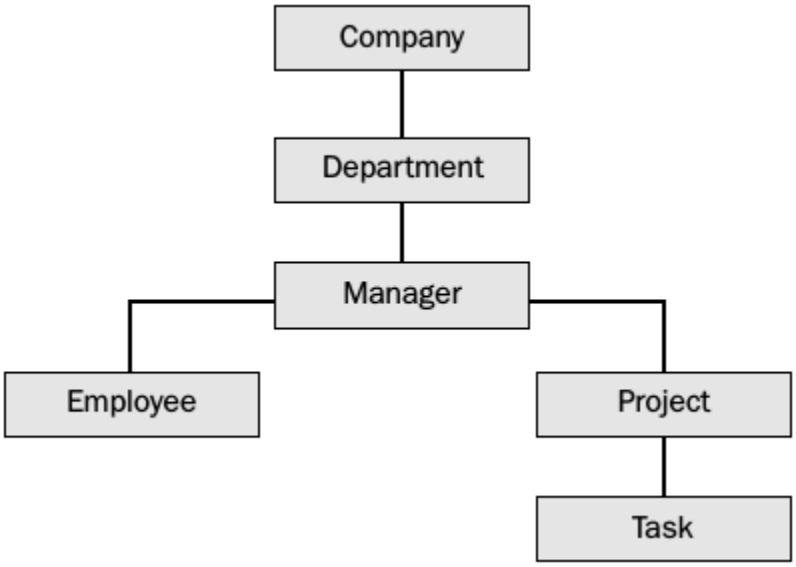
Cualquier acceso a los datos (lectura, escritura y/o, validaciones debe ser explícitamente codificada. Los datos pueden ser almacenados en varios archivos y al igual que las búsquedas cualquier relación entre los datos y/o validaciones debe ser programada.

La capacidad de crecimiento de estos sistemas es limitada.

Jerárquico (Hierarchical)

Este modelo de base de datos, presenta una estructura de árbol invertida. Las tablas de este modelo establecen una relación padre-hijo donde cada tabla hijo puede tener un único padre y cada tabla padre puede tener múltiples hijos.

La mejora presentada respecto del modelo de sistema de archivos es que se establecen relaciones uno a muchos (1 - N)



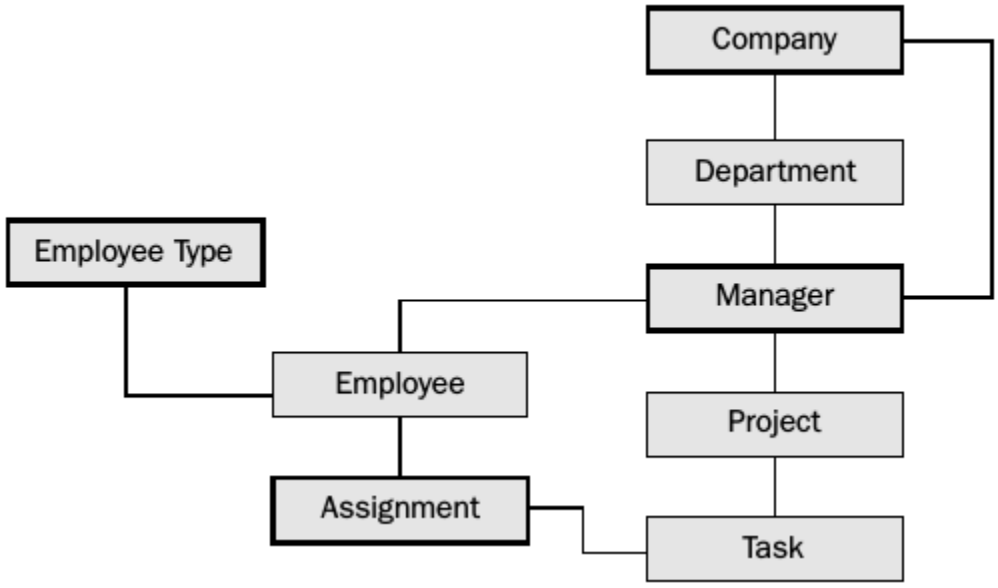
En el ejemplo cada tarea es parte de un proyecto, que a su vez pertenece a un administrador que es parte de un departamento.

Algunas desventajas son que para acceder a un dato hay que conocer toda la estructura, y otra es la rigidez del modelo ya que el hecho de tener que agregar una tabla puede ser un trabajo tedioso si no se tuvo en cuenta al momento del diseño inicial.

Red (Network)

Este modelo es básicamente un refinamiento sobre el modelo jerárquico. En este modelo se permite que una tabla hijo tenga más de una tabla padre.

La mejora presentada respecto del modelo de sistema jerárquico es que se permite establecer relaciones muchos a muchos (N - M)



En el ejemplo se puede ver como los *administradores* pueden ser parte tanto de una *compañía* como de un *departamento*.

Vemos que se agrega la tabla *tipos de empleados* y *asignaciones* para poder definir las asignaciones de cada empleado.

Más allá de tener la posibilidad de definir relaciones N-M, necesitamos definir una única combinación entre padres e hijos, eso se resuelve agregando una tabla intermedia. En el ejemplo la tabla *asignaciones* se agrega para resolver las múltiples combinaciones entre *empleados* y *tareas.* Sin esta tabla intermedia sería imposible encontrar una única asignación de tareas.

Si bien este modelo es más flexible que el jerárquico posee una de las desventajas del mismo que es tener que conocer la estructura completa del modelo para poder acceder a los datos.

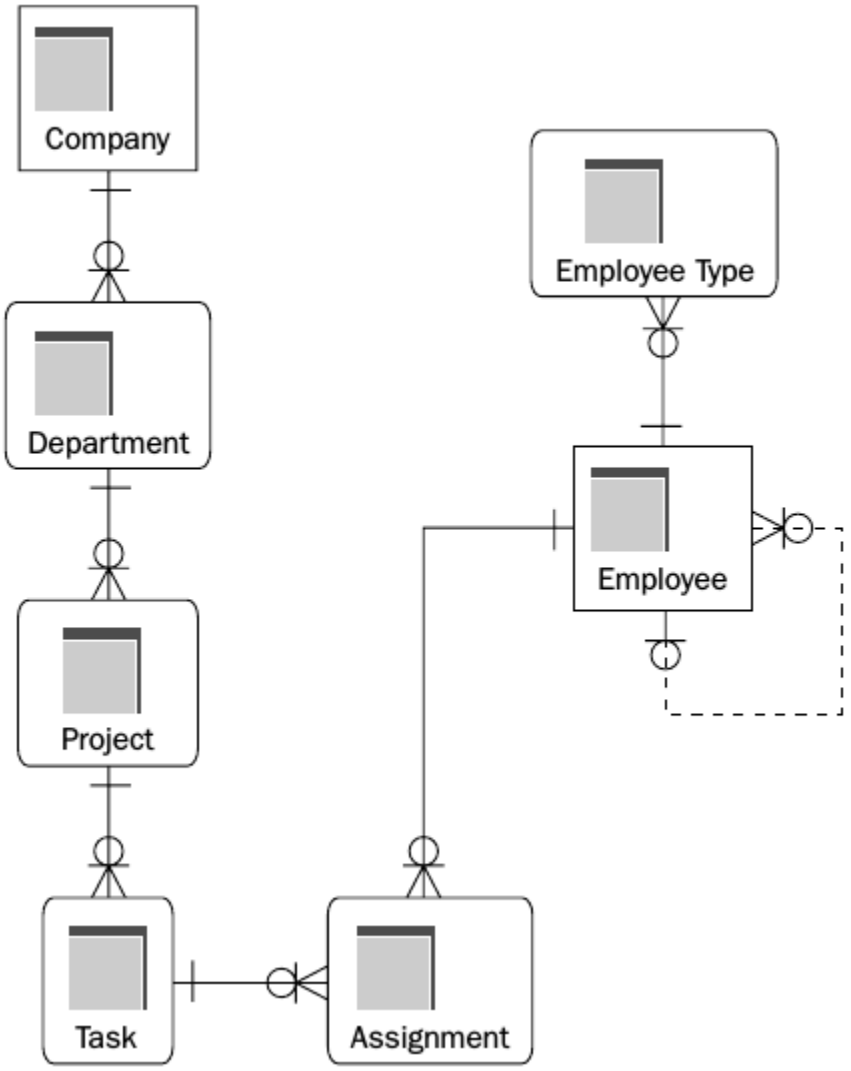
Relacional (Relational)

El modelo relacional, surge para mejorar las restricciones de los modelos anteriores, sin abandonar completamente la estructura jerárquica.

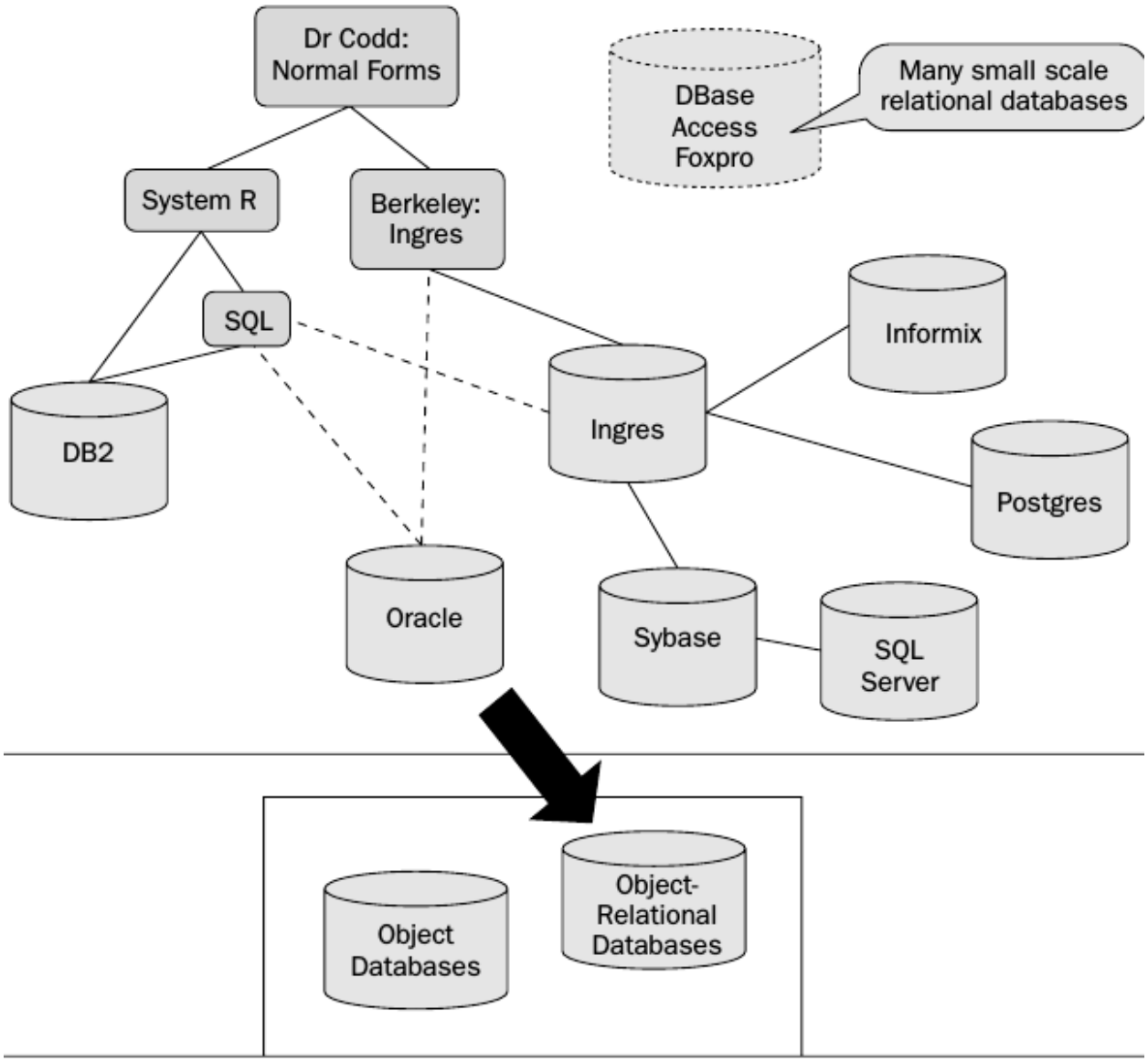
Cualquier dato puede ser accedido sin necesidad de recorrer la estructura completa de tablas, sólo basta con saber qué se está buscando.

Otro de los beneficios es que las relaciones entre tablas no tienen en cuenta su posición jerárquica, cualquier tabla puede ser relacionada con otra siempre y cuando se cuente con los datos apropiados.

Este modelo es más flexible que los anteriores permitiendo agregar tablas y relaciones en cualquier momento sin la complejidad de reestructurar todo el modelo.



En el ejemplo vemos el modelo de datos de proyecto y tareas llevado a un modelo relacional. Se puede ver la relación 1 - N entre Proyecto y Tareas y como la relación entre Empleados y Tareas (N - M) se resuelve con la tabla de asignaciones realizando 2 relaciones 1 - N.



Historia de las bases de datos relacionales

Objeto (Object)

El modelo de datos orientado a objetos es una adaptación para los sistemas de bases de datos del paradigma de la programación orientada a objetos. Estos sistemas son eficientes para acceder a un único dato, pero ineficientes a la hora de devolver varios datos.

Estos modelos resuelven la complejidad de una relación N - M, de las bases de datos relacionales, utilizando colecciones

Objeto-Relacional (Object-Relational)

El modelo objeto-relacional trata de unificar los beneficios de ambos modelos. Básicamente lo que se trata de hacer es incluir las capacidades de un modelo orientado a objetos dentro de un sistema de bases de datos relacional.

NoSQL

Las modelos de bases de datos NoSQL, también llamado “no relacional”, existen desde la década de los 60, pero comenzaron a ser masivamente utilizadas con el advenimiento de la web 2.0. Algunas características que motivó el uso de estos modelos es la flexibilidad en la definición de las estructuras de datos, la velocidad de acceso.

Los motivos para su tardía adopción se deben a que muchos de estos modelos carecen de mecanismos de control de integridad, control de transacciones en beneficio de la velocidad de acceso de los datos.

Tipos de base de datos

Cuando diseñamos y desarrollamos una aplicación, el objetivo principal es proveer una interfaz amigable para el usuario final. Los usuarios definen requerimientos para la aplicación que necesitan y muchos de estos definirán restricciones que derivarán en el tipo de base de datos a utilizar.

A nivel funcional los tipos de base de datos pueden ser:

- Transaccionales
- Sistemas para la toma de decisión (DSS)
- Híbridos

Transaccionales

También conocidas como OLTP (online transaction processing), son bases de datos que basan su operatoria en pequeños cambios a la base de datos. Lo que se trata de hacer es que las operaciones sean atómicas (transacciones) para poder mantener la consistencia de los datos. La función principal de la base de datos es agregar, modificar o eliminar pequeñas cantidades de datos, es decir registros.

Sistemas para la toma de decisión

También conocidos como OLAP (online analytical processing), son sistemas de bases de datos que dan soporte a los altos mandos de una organización para tomar una mejor decisión fundamentada en la información de sus sistemas. En esta categoría se encuentran los sistemas: *data warehouse* que toman como entrada millones de datos históricos de un sistema OLTP (datos que se copian desde el sistema OLTP) para producir información estadística sobre alún aspecto.

Híbridos

Un sistema híbrido, es una mezcla de un sistema OLTP con un sistema *data warehouse*. En casos donde el volúmen de datos no es muy grande puede ser una buena opción costo-beneficio.

OLTP vs OLAP

	OLTP	**OLAP**
Fuente de datos	Datos de operaciones. Provienen de las aplicaciones de usuario	Datos consolidados. Provienen de varios sistemas OLTP
Propósito de los datos	Tareas operativas y de control	Planificación, toma de decisión
Qué significan/muestran los datos	Una foto del estado del negocio	Son múltiples vistas que sobre las distintas actividades del negocio
Actualización de datos	Operaciones cortas y rápidas	Operaciones periódicas y por lote
Consultas	Relativamente simples y rápidas devolviendo pocos registros	A menudo son consultas complejas que incluyen operaciones de consolidación de datos
Velocidad de procesamiento	Operaciones muy rápidas	Depende de la cantidad de datos involucrados, por lo general los procesos en lote demoran horas
Almacenamiento	Con un buen manejo de datos históricos podría no requerir mucho espacio	Requiere mucho espacio de almacenamiento debido a la existencia de datos históricos, índices, etc.
Diseño de la base de datos	Datos Normalizados	Por lo general los datos se encuentran desnormalizados

Backup y recupero	Deben realizarse con mucha frecuencia. Una pérdida de los datos operacionales puede ocasionar pérdidas económicas, problemas legales, etc.	Políticas menos rigurosas que los sistemas OLTP, ya que los datos pueden volver a cargarse.

Conclusión

Las aplicaciones suelen tener una mezcla de necesidades de procesamiento de transacciones y ayuda a la toma de decisiones. Por tanto, el mejor sistema de bases de datos para cada aplicación depende de la mezcla de las dos necesidades que tenga la aplicación.

Bibliografía

- Gavin Powell. Beginning Database Design. Indianapolis: Wiley Publishing, Inc, 2006. ISBN-13: 978-0-7645-7490-0.
- Abraham Silberschatz; Henry F. Korth; S. Sudarshan. FUNDAMENTOS DE BASES DE DATOS. Cuarta Edición. España: McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U, 2002.
- C. J. Date. Introducción A Los Sistemas De Bases De Datos. Séptima Edición. Méjico: Pearson Educación, 2001. ISBN: 968-444-419-2

Diseño y Administración de Base de Datos

Unidad II

Modelo jerárquico

Ing. Alejandro Mostovoi
CUVL - 2018

Índice

Definición de un sistema de base de datos jerárquico **2**

Funcionamiento **2**

Desventajas **3**

Duplicidad de registros 3

Integridad referencial 3

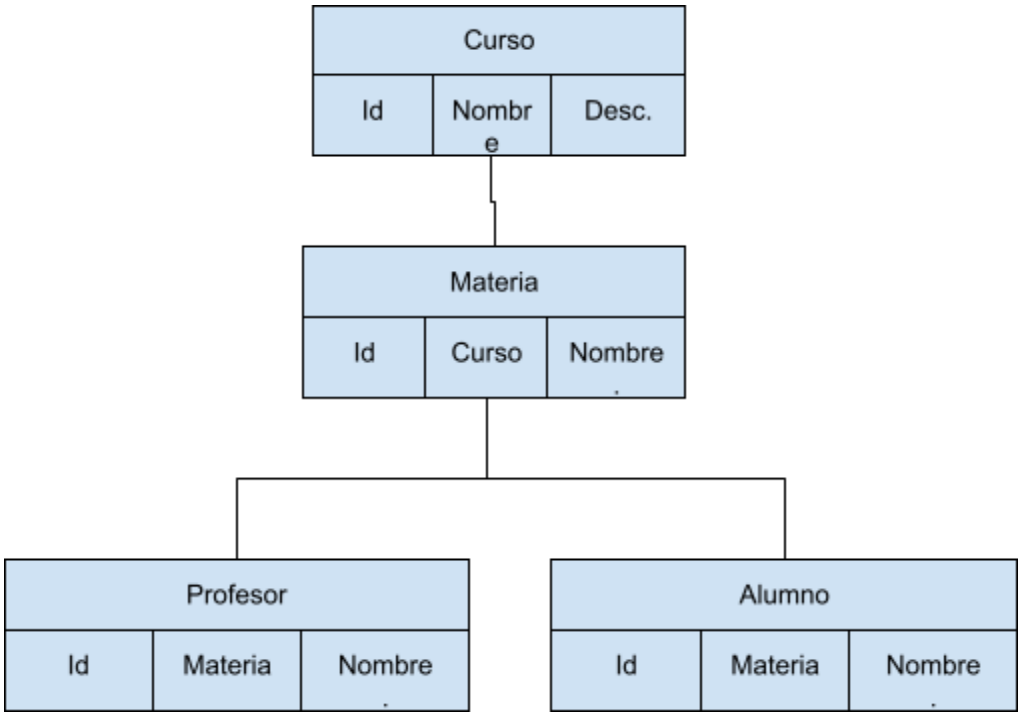
Desnormalización 3

Bibliografía **4**

Definición de un sistema de base de datos jerárquico

Definición	Un sistema de base de datos jerárquico, es aquel que guarda la información en una estructura jerárquica

Los sistemas jerárquicos enlazan los registros en forma de árbol invertido donde un nodo padre puede tener varios nodos hijos, pero un nodo hijo sólo puede tener un único padre.



Funcionamiento

Este modelo permite implementar relaciones *padre-hijo*, que es una mejora presentada frente a los sistemas basados en archivos. Este tipo de relaciones, también denominadas 1-N, son unidireccionales, lo que implica que las relaciones pueden consultarse desde los nodos hijos hacia los nodos padre pero no a la inversa.

La consulta inversa debe realizarse en modo secuencial. También es de destacar que estos modelos, no presentan la posibilidad de crear índices que mejoren las lecturas, como por ejemplo la secuencial.

A diferencia del modelo relacional, las relaciones entre datos se presentan a nivel físico y no lógico, es decir a nivel de direcciones de memoria. Esto significa que un registro hijo posee entre sus datos (metadatos) la dirección de memoria de su registro padre. Esto último representa una ventaja sobre las bases de datos relacionales ya que el acceso de un registro a otro es inmediato.

Desventajas

Los sistemas de bases de datos jerárquicos no implementan mecanismos de control sobre los datos que almacenan, por lo que dichos controles quedan librados a la implementación de la aplicación que vaya a utilizar un motor de base de datos de estas características. A continuación se listan las limitaciones de este tipo de modelo:

Duplicidad de registros

Los sistemas de bases de datos jerárquicos no garantizan la inexistencia de registros duplicados. Es decir, no es posible garantizar que dos registros cualesquiera tengan diferentes valores en un subconjunto concreto de campos. Esto en parte se debe a que este tipo de modelo no implementa el concepto de “clave primaria” ni tampoco el de “campos únicos”.

Integridad referencial

No existe garantía de que un registro *hijo* esté relacionado con un registro *padre* válido. Por ejemplo, es posible borrar un nodo *padre* sin eliminar antes los nodos *hijo*, de manera que éstos últimos están relacionados con un registro inválido o inexistente..

Desnormalización

A diferencia de las bases de datos relacionales, las bases de datos jerárquicas no tienen controles que impidan la desnormalización de una base de datos. Por ejemplo, no existe el concepto de campos clave o campos únicos.

El problema principal de la desnormalización de un modelo es que permite que exista redundancia de datos

Mantenibilidad

Como se mencionó anteriormente, este modelo es difícil de mantener en el sentido de que si durante el ciclo de vida del desarrollo de la aplicación quisiéramos implementar una nueva relación, probablemente todo el modelo se vería afectado.

Bibliografía

- Gavin Powell. Beginning Database Design. Indianapolis: Wiley Publishing, Inc, 2006. ISBN-13: 978-0-7645-7490-0.
- Abraham Silberschatz; Henry F. Korth; S. Sudarshan. FUNDAMENTOS DE BASES DE DATOS. Cuarta Edición. España: McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U, 2002.
- C. J. Date. Introducción A Los Sistemas De Bases De Datos. Séptima Edición. Méjico: Pearson Educación, 2001. ISBN: 968-444-419-2
- Base de datos jerárquica. Wikipedia, La enciclopedia libre, 2018 [consulta: 06 de marzo del 2018]. Disponible en https://es.wikipedia.org/wiki/Base_de_datos_jer%C3%A1rquica
- Understanding the Hierarchical Database Model. Disponible en https://mariadb.com/kb/en/library/understanding-the-hierarchical-database-model/

Diseño y Administración de Base de Datos

Unidad III

Modelo de red

Ing. Alejandro Mostovoi

CUVL - 2018

Índice

Definición de un sistema de base de datos de red **2**

Funcionamiento **2**

Ventajas **3**

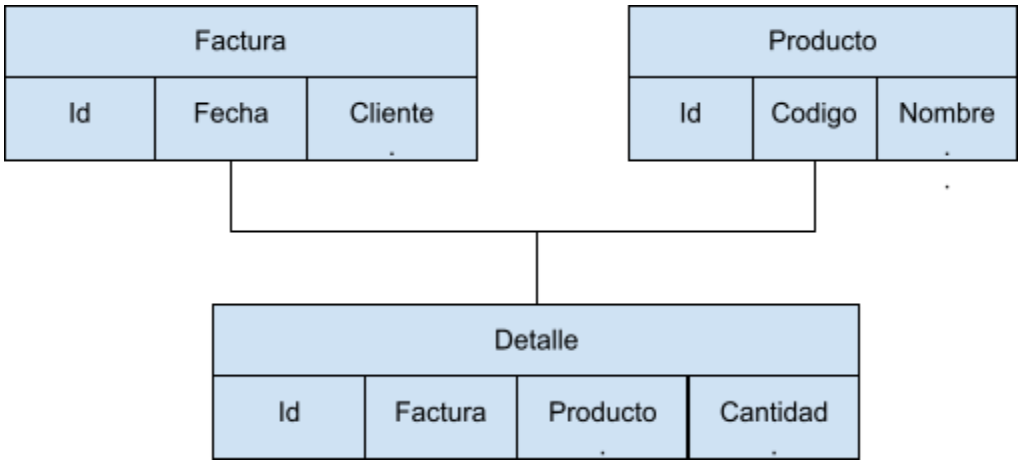
Desventajas **3**

Bibliografía **4**

Definición de un sistema de base de datos de red

Definición	Un sistema de base de datos de red, es aquel que guarda una colección o *set* de registros, conectados entre sí por medio de enlaces como una red.

Una de las características distintivas de este modelo es que permite representar las relaciones de forma más natural y permite que un nodo hijo posea más de un nodo padre a diferencia del modelo jerárquico.



Funcionamiento

Si bien este modelo permite implementar relaciones más complejas que el modelo jerárquico, también se utilizan listas enlazadas para navegar los datos.

Las relaciones o conjuntos de registros son denominadas *sets*. Estos *sets* al igual que los registros deben ser descritos en el schema de la base de datos.

Un *set* de registros tiene las siguientes características:

- Incluye un único registro de un tipo. Este registro es denominado *owner* (dueño) de la relación
- Incluye 0 (cero), 1 (uno) o más registros de un segundo tipo. Estos registros son denominados (members) miembros de la relación
- Los sets están ordenados.

Ventajas

- Permite manejar más tipos de relaciones que el modelo jerárquico
- Es más flexible que el modelo jerárquico, aunque no todos los problemas se pueden resolver agregando “padres” a las relaciones.

Desventajas

- La navegación de los datos sigue siendo un problema al igual que en el modelo jerárquico. Hay que conocer todo el modelo
- La curva de aprendizaje del lenguaje para consultar los datos es muy grande
- Los cambios estructurales son todavía un problema debido a que la implementación de las relaciones es física y no lógica.
- Un cambio en la estructura de las relaciones demandará un cambio en el código dela aplicación

Bibliografía

- Gavin Powell. Beginning Database Design. Indianapolis: Wiley Publishing, Inc, 2006. ISBN-13: 978-0-7645-7490-0.
- Abraham Silberschatz; Henry F. Korth; S. Sudarshan. FUNDAMENTOS DE BASES DE DATOS. Cuarta Edición. España: McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U, 2002.
- C. J. Date. Introducción A Los Sistemas De Bases De Datos. Séptima Edición. Méjico: Pearson Educación, 2001. ISBN: 968-444-419-2
- Network model. Wikipedia, La enciclopedia libre, 2018 [consulta: 06 de marzo del 2018]. Disponible en https://en.wikipedia.org/wiki/Network_model
- 1. Network (CODASYL) Data Model. Disponible en: https://coronet.iicm.tugraz.at/wbtmaster/allcoursescontent/netlib/ndm1.htm

Diseño y Administración de Base de Datos

Unidad IV

Bases de datos relacionales

Ing. Alejandro Mostovoi

Año - 2023

Revisión: 1.1

Tabla de versiones

Autor	Fecha	Versión	Descripción
Alejandro Mostovoi	21/04/18	1.0	Versión Inicial
Alejandro Mostovoi	28/08/23	1.1	Corrige definición de DML

Índice

Arquitectura de los sistemas de bases de datos **3**
- Nivel externo 3
- Nivel conceptual 4
- Nivel interno 4

El modelo relacional **5**
- Conceptos básicos 5
- Relaciones 6
  - Restricciones sobre las relaciones. 8
- Claves 9
  - Clave Primaria (Primary Key) 9
  - Clave Única (Unique Key) 9
  - Clave Foránea (Foreign Key) 10
- Integridad referencial 10
- Índices 11
  - Qué es un índice 11
  - Cómo funciona un índice 11
  - Tipos de índices 11

Diagrama Entidad-Relación **13**

SQL - DDL **13**
- Creación de Tablas 13
  - Campos y restricciones sobre campos 14
- Modificación de la estructura de una tabla 16
- Índices 17
- Eliminar una tabla 17

SQL - DML **18**
- Insertar datos 18
- Actualizar datos 19
- Eliminar datos 19
- Consultar datos 19
- Funciones de agregado 22

Bibliografía **24**

Arquitectura de los sistemas de bases de datos

Una de las arquitecturas más utilizadas por los sistemas de bases de datos es la arquitectura ANSI/SPARC, que presenta 3 niveles conocidos como *interno, conceptual* y *externo.*

- El nivel externo (también conocido como lógico de usuario) es el más próximo a los usuarios; es decir, el que tiene que ver con la forma en que los usuarios individuales ven los datos.
- El nivel conceptual (también conocido como el lógico, sin calificar) es un nivel de indirección entre los otros dos.
- El nivel interno (también conocido como físico) es el que está más cerca del almacenamiento físico; es decir, es el que tiene que ver con la forma en que los datos están almacenados físicamente.

El nivel conceptual, es único. Si bien pueden existir muchas vistas del nivel externo, el nivel conceptual es único, es el modelo lógico de cómo se organizan y relacionan los datos dentro de la base de datos.

Por ejemplo:

Se tiene una base de datos que guarda datos de empleados de una organización.

En el nivel interno de la base de datos los empleados están representados por un registro de 20 bytes de longitud. Este registro posee 4 campos de la siguiente manera: los primeros 6 bytes corresponden al código de empleado, los próximos 4 bytes pertenecen al código de departamento, mientras que los últimos 10 bytes se corresponden con el salario del empleado.

En el nivel conceptual, la base de datos contiene información relacionada a una entidad llamada *Empleado* donde cada empleado posee un código (6 caracteres), un departamento (4 caracteres) y un salario (5 dígitos decimales).

Por último en el nivel externo existen 2 vistas una para usuarios de tesorería que pueden ver todos los datos de un empleado y otra vista para usuarios del departamento de RRHH que sólo pueden ver el código de empleado y departamento al que pertenecen.

Nivel externo

Este nivel es el del usuario que bien puede ser un usuario final, un desarrollador de aplicaciones o un DBA (database administrator).

Para un usuario final, este nivel expone un lenguaje de consulta o bien una aplicación de finalidad específica que permite crear una consulta a través de elementos gráficos.

Para un usuario desarrollador, este nivel tendrá un lenguaje de programación convencional (Java, C++, PHP, etc.) con interfaces para interactuar con la base de datos. En principio estas interfaces serán una combinación de dos lenguajes ya mencionados que son DDL y DML.

- DDL consiste en aquellas construcciones declarativas necesarias para declarar objetos de base de datos (Tablas, campos, índices, restricciones, vistas, funciones).
- DML consiste en aquellas instrucciones ejecutables que transfieren información hacia y desde la base de datos (consultas, altas, bajas y modificaciones de datos)

Volviendo a la arquitectura, la vista externa o nivel externo es para un usuario final la base de datos propiamente dicha.

Cada vista externa, está definida mediante un **esquema conceptual** que consiste básicamente en definiciones de cada uno de los objetos de la base de datos.

Nivel conceptual

La vista conceptual está definida por medio del esquema conceptual, el cual comprende definiciones de cada uno de los diversos tipos de registros conceptuales.

Para definir el esquema conceptual se utiliza el lenguaje de definición de datos DDL. Desde un punto de vista teórico y estricto, el esquema conceptual, no debe tener ninguna referencia sobre la representación física de los campos, formas de almacenamiento, restricciones de seguridad o integridad.

En resumen, el esquema conceptual es una forma de describir qué datos se almacenan dentro de la base de datos y sus relaciones

Nivel interno

Este nivel es una representación de bajo nivel de toda la base de datos. La vista interna, está por delante del nivel físico, ya que no tiene que ver con cómo se definen las páginas de memoria o el tamaño de los cilindros de disco, etc.

La vista interna, sólo define los diversos tipos de registros, índices, etc.

El modelo relacional

A modo de introducción, podemos decir que el modelo de datos entidad-relación (E-R) percibe el mundo real como un conjunto de objetos llamados entidades y de relaciones entre dichos objetos.

Es un sistema donde el usuario percibe la información como tablas, su aspecto estructural; estas tablas satisfacen ciertas restricciones de integridad, su aspecto de integridad; proporciona operadores para manipular los datos, son operadores que operan sobre tablas y su resultado deriva en tablas, aspecto de manipulación.

Conceptos básicos

Entidad: es un objeto en el mundo real que es distinguible de todos los demás objetos. Por ejemplo una persona, una cuenta bancaria, un libro. Una entidad posee un conjunto de propiedades o atributos que la describen y que también permiten identificarla de forma unívoca.

Cada uno de los atributos de una entidad posee un conjunto posible de valores que puede tomar que es llamado **dominio.** Por ejemplo, el dominio del atributo *edad* de una entidad *Persona* podría ser los valores enteros entre 0 y 130.

Dentro de una base de datos relacional, una entidad está representada por un registro y sus atributos por campos. El conjunto de entidades del mismo tipo se agrupan dentro de una tabla.

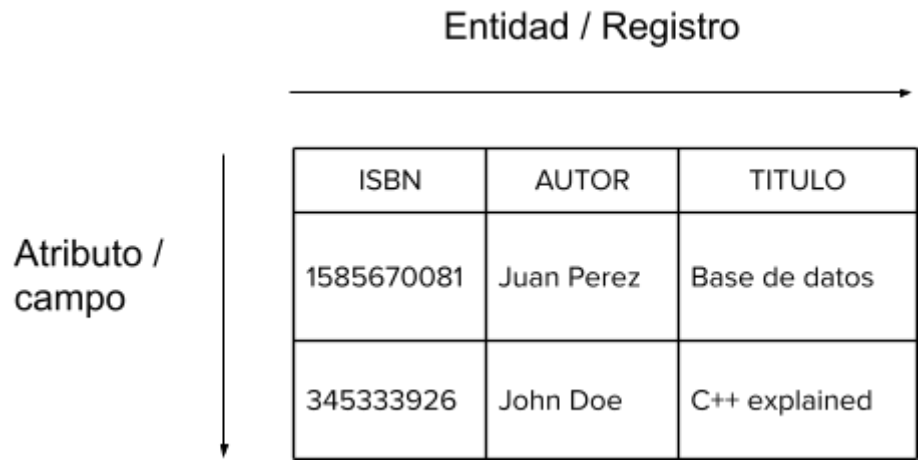
Registro: el término registro, **fila** o **tupla** todos significan lo mismo. Son términos usados para describir una entidad, un conjunto de atributos, dentro de una tabla.

Campo: el término campo, **columna** o **atributo** todos significan lo mismo. Son términos usados para describir un campo en una tabla. Los campos se repiten en cada registro (entidad), pero no necesariamente sus valores van a repetirse entre una entidad y otra.

Tabla: es la representación de un conjunto de entidades del mismo tipo dentro de una base de datos relacional. Es una estructura lógica. Las tablas son una abstracción de cómo se almacenan los datos físicamente.

Como mencionamos anteriormente, es una estructura de datos compuesta por campos y registros donde todos los registros poseen los mismo campos (no valores), es decir comparten la misma estructura.

En la introducción comentamos que los operadores actúan sobre tablas y su resultado es otra tabla, esto es a nivel conceptual, no significa que el resultado debe materializarse en una tabla en la base de datos. Es importante destacar de este aspecto que como todos los operadores operan sobre una tabla y su resultado es otra tabla, entonces podemos decir que el resultado de una operación puede ser la entrada de otra y esto nos permite anidar operaciones



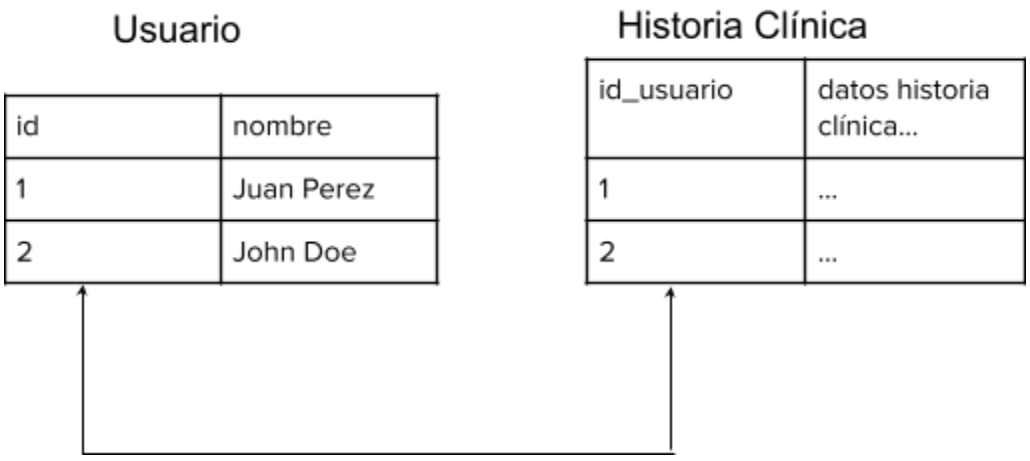
Relaciones

Una relación es una asociación entre diferentes entidades. Por ejemplo, se podría asociar una entidad *libro* con una entidad *autor*.

La cantidad de tablas que intervienen en una relación es el grado de la misma, entonces si en una relación intervienen 2 tablas, será una relación binaria, si intervienen 3 tablas será ternaria y así sucesivamente. Lo más común es que el grado sea binario.

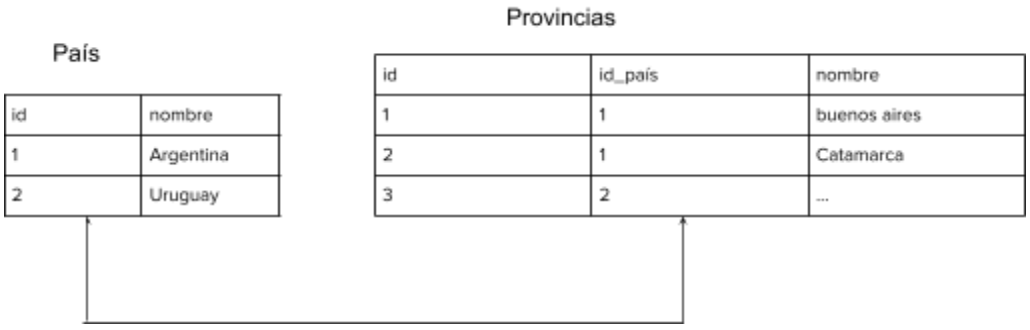
Además del grado, las relaciones poseen otro atributo que es la cardinalidad. Este atributo expresa el número de entidades a las que otra entidad puede estar asociada. Los tipos de cardinalidad más comunes son:

- Uno a Uno (1 - 1): una entidad en A se asocia con a lo sumo una entidad en B y viceversa. Por ejemplo, una persona y su historia clínica.



Las relaciones 1 a 1 son poco comunes, son utilizadas para eliminar repetidos valores NULL en campos.

- Uno a muchos (1 - N): una entidad en A se asocia con uno o más en la entidad B pero una entidad en B se asocia sólo con 1 entidad en A. Por ejemplo un país y sus provincias.



Las relaciones 1 - N son las más comunes en las bases de datos relacionales.

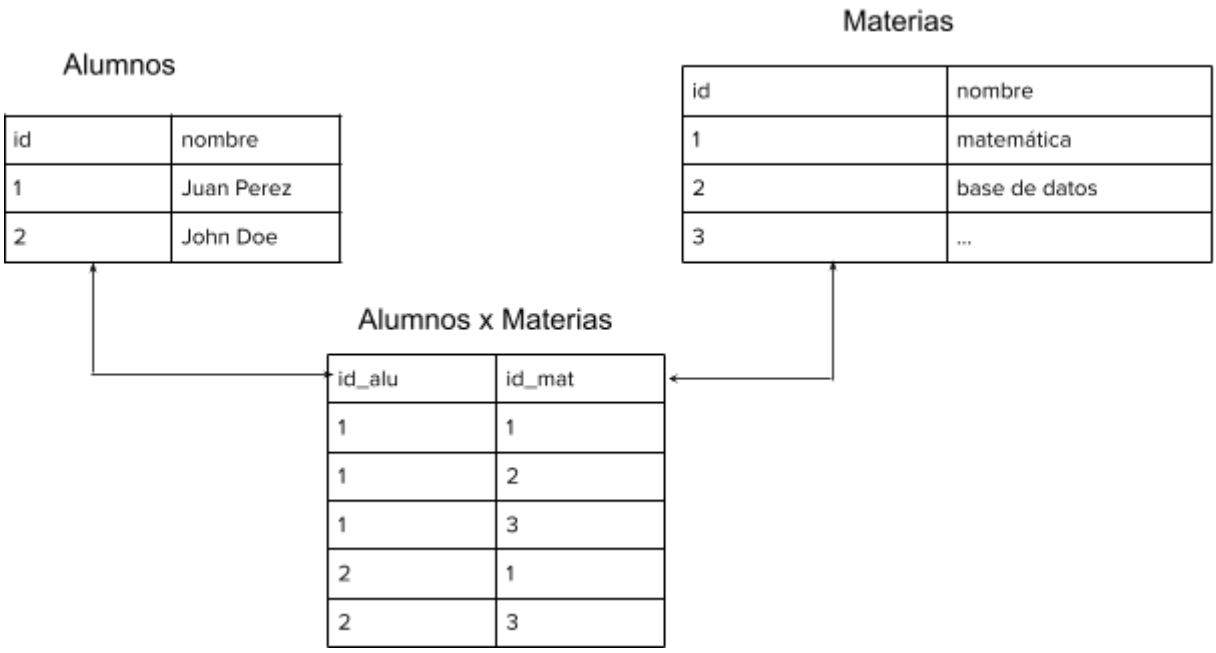
- Muchos a muchos (N - M): una entidad en A se asocia con uno o más en la entidad B y una entidad en B se asocia con uno o más entidades en A. Por ejemplo alumnos que pueden inscribirse a muchas materias.

Si bien este tipo de relaciones se puede resolver en una única tabla, posee el problema de la redundancia de datos y aún más importante se hace costoso obtener un único alumno o una única materia.

Datos

nombre	materia
Juan Perez	matemática
Juan Perez	base de datos
John Doe	base de datos
John Doe	...

Para resolver estos problemas, se debe crear una tabla intermedia y 2 relaciones 1 - N.



Restricciones sobre las relaciones.

Las relaciones entre entidades pueden ser obligatorias o únicas (o no repetibles) y combinaciones de éstas.

Obligatorias: implica que el atributo que referencia una entidad en la tabla relacionada debe referenciar de forma obligatoria una entidad en la tabla relacionada. En contrapartida, una relación no obligatoria permite que el atributo de la entidad quede vacío.

Únicas: implica que el atributo que referencia una entidad en la tabla relacionada no puede ser repetido entre entidades de la misma tabla.

Obligatorias y Únicas: implica que el atributo que referencia una entidad en la tabla relacionada no puede repetirse y tampoco puede quedar vacío.

Claves

En una base de datos relacional, la palabra *clave (key)* es un término usado para denotar campos que pueden vincular tablas, es decir, participar de una relación. Una clave puede estar conformada por uno o más campos.

Una clave es además un índice (ver Índices), porque los valores de ese campo son copiados a una estructura de búsqueda más eficiente.

Existen 3 tipos de claves: Primaria, Única y Foránea.

Clave Primaria (Primary Key)

Una clave primaria es utilizada para identificar de forma *unívoca* un registro dentro de una tabla.

Al definir un campo, o conjunto de campos, como clave primaria lo que estamos haciendo es establecer una restricción sobre ese o esos campos donde los mismos no pueden repetirse entre registros ni quedar vacíos, en otras palabras, los campos en cuestión son únicos y obligatorios.

Por lo general y por cuestiones de performance (rendimiento) en las consultas, los campos que conforman una clave primaria serán campos de tipo entero (integer).

A la hora de elegir el o los campos que conformarán la clave primaria, debemos pensar en aquellos campos que identifican de forma unívoca una entidad. Por ejemplo en el caso de *personas* podríamos utilizar el DNI como clave primaria.

Sólo puede definirse una única clave primaria por tabla.

Clave Única (Unique Key)

Al igual que con las claves primarias, una clave única establece una restricción de unicidad sobre el o los campos involucrados. Esto significa que los valores de dichos campos no pueden repetirse entre los registros de la tabla, deben ser únicos.

Distinto a lo que ocurre con las claves primarias, se puede definir tantas claves únicas como se necesiten.

Los campos definidos como *clave única* no son utilizados en relaciones entre tablas, a menos que dicho campo posea a su vez una restricción de obligatoriedad (campo no nulo).

Por ejemplo, en el caso de *usuarios* de un sistema se podría definir una clave única sobre el campo *email*.

Clave Foránea (Foreign Key)

Una *clave foránea* es un campo cuyos valores śerán las copias del campo definido como *clave primaria* en la tabla “padre” de la relación. Una *clave foránea* define la referencia para cada registro dentro de la tabla “hijo” de la relación. hacia los registros de la tabla “padre” de la relación.

Por ejemplo en el caso de tabla *Países* y *Provincias,* en la 2da tabla vamos a tener una columna *pais_id* que será una clave foránea a los registros de la tabla *País*.

Por definición un campo sobre el que se define una clave foránea puede ser vacío o en caso de tener un valor, el mismo debe ser uno de los valores del campo referenciado en la tabla padre.

Al igual que las claves únicas, se pueden definir todas las claves foráneas que se necesiten.

Un campo de una tabla “hijo” sólo puede participar de una clave foránea.

Integridad referencial

Como su nombre lo indica, la integridad referencial, asegura la integridad de las relaciones definidas por claves primarias y foráneas.

Este mecanismo, asegura que si sobre un campo se define una clave primaria, los valores del mismo no se repitan entre los registros de la tabla en cuestión y que no sean vacíos (nulos). En el caso de una relación entre tablas verifica que los valores cargados en un campo sobre el que se definió una clave foránea, existan en el campo referenciado (que por lo general será una clave primaria):

También verifica que si se intenta modificar los datos de un campo que participa en una relación (o eliminar el registro), no queden registros huérfanos en tablas “hijo”.

Índices

Qué es un índice

Un índice, es una estructura que almacena datos copiados del campo de la tabla sobre el cual se definió el índice. Su función, al igual que el índice de un libro, es permitir encontrar algo de forma más eficiente, en un libro será un capítulo del libro mientras que en tabla será un registro de la misma.

La forma de almacenar los índices varía de DBMS a DBMS, lo importante es que todos establecen la separación en el nivel físico.

Cómo funciona un índice

Cuando realizamos una consulta a la base de datos, el *Procesador de Consultas* (el encargado de generar los planes de consultas) define si debe recorrer la tabla completa (escaneo secuencial) o si utilizará alguno de los índices definidos para obtener los registros solicitados de forma más rápida.

No siempre el acceso a través de los índices es el más rápido, por ejemplo si la tabla posee pocos registros siempre será más rápido un acceso secuencial que un acceso a través de alguno de los índices definidos.

Otra cuestión a tener en cuenta es que cada actualización a una tabla (crear, modificar o eliminar un registro) impactará en los índices que la misma posea, por ende una tabla con muchos índices se traducirá en una baja en el rendimiento en las operaciones de alta, baja o modificación de registros.

Tipos de índices

Existen distintos tipos de índices, a continuación mencionamos algunos de los más comunes:

B-Tree

- Ideal sobre columnas que posean valores únicos.
- Puede proveer los datos ordenados
- Mejor performance en temas de concurrencia

Generalized Search Tree (GIST)

- Ideal para indexar datos complejos que no se “adaptan” a un índice b-tree
- Permite indexar datos geográficos, arrays, par clave/valor

Generalized Inverted Index (GIN)

- Ideal para indexar valores compuestos
- Ideal para indexar columnas con muchos duplicados, ya que por cada clave puede tener punteros a varias filas. Esto también reduce el tamaño del índice
- Este índice guarda un conjunto de pares (clave, valor) donde el “valor” es un conjunto de IDs de filas donde ocurre esa clave
- Muy utilizado en valores de tipo: documento, JSON, arrays

Otros

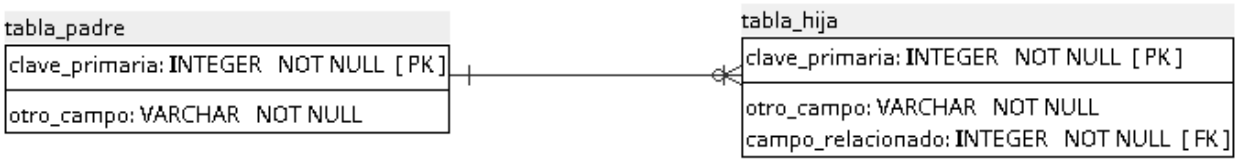
- Índices por expresiones (Expression Index)
    - Permite definir el índice a partir de una expresión, función, concatenación, etc.
    - Ej: lower(nombre)
- Índices parciales (Partial Index)
    - Permite definir un índice para un subconjunto de filas y no toda la tabla
    - Reduce el overhead en operaciones de insert/update
    - Reduce el tamaño del índice
    - Ej: CREATE INDEX state_idx ON customer (state) WHERE state = ’AZ’;

Resumen

- Sólo B-tree, GIN y GIST soportan índices multicolumna
- B-tree ideal sobre columnas que posean valores únicos.
- GIN Ideal para indexar columnas con muchos duplicados
- GIST para todo lo demás

Diagrama Entidad-Relación

Ejemplo de relación 1 - 1



Ejemplo de relación 1 - N

SQL - DDL

El “sublenguaje” DDL (Data Definition Language) es el conjunto de sentencias que permite administrar las estructuras de la base de datos. Se compone de instrucciones que permiten crear, modificar y eliminar los objetos de una base de datos relacional como ser: tablas, campos, índices, vistas, etc.

Creación de Tablas

Como mencionamos anteriormente una tabla es una estructura lógica, compuesta por columnas (campos) y filas (registros), donde cada campo posee un nombre y tipo de dato específico.

PostgreSQL nos presenta el comando ***CREATE TABLE*** que nos permite crear tablas.

La sintaxis básica de este comando sería:

```
CREATE TABLE <nombre_de_tabla> (
    <nombre_de_campo> <tipo> <restricciones sobre el campo>,
    <nombre_de_campo_2> <tipo> <restricciones sobre el campo>,
    ...
    <nombre_de_campo_N> <tipo> <restricciones sobre el campo>
```

```
    <restricciones sobre la tabla>
);
```

Un ejemplo simple de este comando sería:

```
CREATE TABLE my_first_table (
    first_column text,
    second_column integer
);
```

Campos y restricciones sobre campos

Cómo se vió en el ejemplo anterior, ll comando ***CREATE TABLE*** también nos permite especificar los campos y sus restricciones.

La sintaxis básica para agregar un campo es:

```
<nombre_del_campo> <tipo_de_dato> <restricciones>,
```

Los tipos de datos más comunes son:

Numéricos	Texto	Date	Boolean
• smallint • smallserial • integer • serial • bigint • bigserial • double	• text • character • character varying	• date • timestamp	• boolean

Las restricciones más comunes sobre los campos son:

- NOT NULL: especifica que el valor de dicho campo no puede ser null, es decir, el campo es obligatorio.
- UNIQUE: especifica que el valor de dicho campo no puede repetirse entre registros
- PRIMARY KEY: especifica que el valor de dicho campo es obligatorio y no puede repetirse entre registros

- CHECK: permite realizar validaciones sobre el campo
- REFERENCES: especifica que dicho campo es una clave foránea.

Ejemplos

```
CREATE TABLE usuarios (
    id bigserial PRIMARY KEY,
    name character varying (30) NOT NULL,
    surname character varying (30) NOT NULL,
    email character varying (30) NOT NULL UNIQUE,
    age smallint CHECK (age > 0),
    type char DEFAULT ('C')
);
```

```
CREATE TABLE telefonos (
    id serial CONSTRAINT tel_pk PRIMARY KEY,
    id_usuario bigint REFERENCES usuarios (id),
    telefono_1 character varying (15),
    telefono_2 character varying (15),
    telefono_3 character varying (15)
);
```

```
CREATE TABLE direcciones (
    id serial,
    id_usuario bigint,
    direccion_1 character varying (50),
    direccion_2 character varying (50),

    CONSTRAINT dir_pk PRIMARY KEY(id),
    CONSTRAINT usr_fk FOREIGN KEY (id_usuario) REFERENCES usuarios (id)
);
```

Para una definición completa de este comando ver:

https://www.postgresql.org/docs/current/static/sql-createtable.html

Modificación de la estructura de una tabla

En caso de que necesitemos modificar la estructura de una tabla ya creada, debemos utilizar el comando ***ALTER TABLE***. Este comando nos permitirá renombrar la tabla y sus campos, cambiar el tipo de datos de los campos (siempre que no existan conflictos con los datos o relaciones), agregar o eliminar campos, etc.

Sintaxis básica:

```
alter table <nombre_de_tabla> <acción>;
```

Ejemplos:

Agregar una columna

```
alter table usuarios add column description character varying (50);
```

Eliminar una columna

```
alter table usuarios drop column description;
```

Renombrar una columna

```
alter table usuarios rename column description to comments;
```

Cambiar el tipo de dato de una columna

```
alter table usuarios alter column age type integer;
```

Agregar una restricción a un campo

```
alter table usuarios alter column comments SET NOT NULL;
```

Eliminar una restricción a un campo

```
alter table usuarios alter column comments DROP NOT NULL;
```

Para una definición completa de este comando ver:

https://www.postgresql.org/docs/current/static/sql-altertable.html

Índices

Como se mencionó anteriormente, un índice es una estructura ordenada que facilita el acceso rápido (en la mayoría de los casos) a los datos de una tabla.

PostgreSQL soporta varios tipos de índices: B-tree, Hash, GiST, SP-GiST y GIN. Tener en cuenta que cada uno utiliza un algoritmo de búsqueda diferente.

Por defecto, el comando ***CREATE INDEX*** crea índices de tipo B-tree ya que es el que mejor se adecúa a la mayoría de los casos.

Sintaxis básica

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table [ USING method
]( { column | ( expression ) }[ ASC | DESC ] [ NULLS { FIRST | LAST } ] [,
...] )[WHERE predicate]
```

- Concurrently: permite crear el índice sin bloquear la tabla
- Method: uno de btree, hash, gin, gist, spgist, brin. Si no se especifica asume btree
- Expression: permite definir una expresión para crear el índice.
- Predicate: expresión para definir un índice parcial.

Siempre que se cree un índice sobre una tabla con datos y también periódicamente se debe ejecutar el comando:

```
VACUUM FULL ANALYZE [VERBOSE] <nombre_de_tabla>;
```

Este comando realiza un mantenimiento de los índices sobre la tabla y sobre sus datos.

Eliminar una tabla

```
drop table <nombre_de_tabla>;
```

SQL - DML

El “sublenguaje” DML (Data Manipulation Language) es el conjunto de sentencias que permite administrar las estructuras de la base de datos. Se compone de instrucciones que permiten crear, modificar y eliminar entidades en una base de datos relacional .

Insertar datos

Para insertar datos en una tabla, contamos con el comando ***INSERT INTO***. Este comando permite insertar 1 registro en una tabla, especificando los campos afectados y sus valores.

Sintaxis básica

```
insert into <tabla> (<campo_1>, <campo_2>, ..., <campo_n>) values
(<valor_campo_1>, <valor_campo_2>, ..., <valor_campo_n>);
```

Ejemplos:

```
insert into usuarios (name, surname, age) values (‘Juan’, ‘Perez’, 20);
```

Tener en cuenta que aquellos campos sobre los que se definió una secuencia (autonumerados) no deberían ser especificados a menos que se les quiera establecer un valor distinto al que asignaría la secuencia.

Los campos que tienen un valor por defecto en caso de que se especifiquen en la sentencia *insert* serán tenidos en cuenta y se insertarán con el valor especificado. Si no se especifican se insertarán con el valor por defecto definido.

Otra forma de realizar un insert es utilizando un query para especificar los valores.

```
insert into <tabla> (<campo_1>, <campo_2>, ..., <campo_n>)

select campo_x, campo_y, ..., campo_z

from <tabla>;
```

Actualizar datos

Para actualizar datos en una tabla, contamos con el comando ***UPDATE***. Este comando permite actualizar 1 o más registros en una tabla.

Sintaxis básica

```
update <tabla> set <campo_1> = <valor_1, <campo_2> = <valor_2>, ...,
<campo_n> = <valor_n>

[where <condición>];
```

Ejemplos:

```
update usuarios set description = 'hola mundo' where id < 10;
```

Eliminar datos

Para eliminar datos en una tabla, contamos con el comando ***DELETE***. Este comando permite eliminar 1 o más registros en una tabla.

Sintaxis básica

```
delete from <tabla> [where <condición>];
```

Ejemplos:

```
delete from usuarios where id < 10;
```

Consultar datos

Para consultar datos en una tabla, contamos con el comando ***SELECT***. Este comando permite consultar 1 o más registros en una tabla o varias tablas.

Sintaxis básica

```
select <campos| *> from <tabla>;
```

Ejemplos:

```
-- devuelve todos los campos de la tabla alumnos.

select * from alumnos;

-- devuelve legajo, apellido y nombre de la tabla alumnos

select legajo, apellido, nombre from alumnos;
```

Sintaxis completa

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]

[ * | expression [ [ AS ] output_name ] [, ...] ]

[ FROM from_item [, ...] ]

[ WHERE condition ]

[ GROUP BY grouping_element [, ...] ]

[ HAVING condition [, ...] ]

[ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]

[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST |
LAST } ]

[, ...] ]

[ LIMIT { count | ALL } ]

[ OFFSET start [ ROW | ROWS ] ]
```

SELECT: Palabra clave que indica que la sentencia de SQL que queremos ejecutar es de selección.

ALL: Indica que queremos seleccionar todos los valores. Es el valor por defecto y no suele especificarse casi nunca.

DISTINCT: Indica que queremos seleccionar sólo los valores distintos.

FROM: Indica la tabla (o tablas) desde la que queremos recuperar los datos. En el caso de que exista más de una tabla se denomina a la consulta "consulta combinada" o "join". En las consultas combinadas es necesario aplicar una condición de combinación a través de una cláusula WHERE.

WHERE: Especifica una condición que debe cumplirse para que los datos sean devueltos por la consulta. Admite los operadores lógicos AND y OR.

GROUP BY: Especifica la agrupación que se da a los datos. Se usa siempre en combinación con funciones agregadas.

HAVING: Especifica una condición que debe cumplirse para que los datos sean devueltos por la consulta. Su funcionamiento es similar al de WHERE pero aplicado al conjunto de resultados devueltos por la consulta. Debe aplicarse siempre junto a GROUP BY y la condición debe estar referida a los campos contenidos en ella.

ORDER BY: Presenta el resultado ordenado por las columnas indicadas. El orden puede expresarse con ASC (orden ascendente) y DESC (orden descendente). El valor predeterminado es ASC.

LIMIT: permite establecer la cantidad de registros a devolver

OFFSET: establece el número de registro a partir del cual se devuelven registros.

LIMIT y OFFSET son utilizados para armar paginado de datos.

Funciones de agregado

Las funciones de agregado realizan un cálculo sobre un conjunto de valores y devuelven un solo valor.

Las funciones de agregado se suelen utilizar con la cláusula GROUP BY de la instrucción SELECT.

Todas las funciones de agregado son deterministas. Esto significa que las funciones de agregado devuelven el mismo valor cada vez que se las llama con un conjunto específico de valores de entrada.

Con las funciones de agregado de SQL, se pueden determinar varias estadísticas relacionadas con conjuntos de valores.

Las funciones de agregado son:

avg: permite obtener el promedio de valores de un campo

count: permite contar valores de un campo. Se puede utilizar en conjunción con la cláusula *distinct* para sólo contar los valores distintos sobre un campo

min: obtiene el mínimo valor de un campo

max: obtiene el máximo valor de un campo

sum: sumariza los valores de un campo

Ejemplos:

Dada la tabla *empleados:*

Legajo	Nombre	Apellido	Departamento	Salario
123	Juan	Perez	Sistemas	40000
456	Ana	Martinez	Sistemas	45000
876	Valeria	Rodriguez	Legales	50000

```
-- Obtener la cantidad de empleados de cada departamento

select departamento, count(*)

from empleados

group by departamento;
```

```
-- Obtener la cantidad de departamentos a partir de la tabla empleados

select count(distinct departamento)

from empleados;
```

```
-- Obtener el salario mínimo, el salario máximo y promedio de los salarios
mayores a 20000 por departamento

select departamento, min(salario), max(salario), avg(salario)

from empleados

group by departamento

having avg(salario) > 20000;
```

Bibliografía

- Gavin Powell. Beginning Database Design. Indianapolis: Wiley Publishing, Inc, 2006. ISBN-13: 978-0-7645-7490-0.
- Abraham Silberschatz; Henry F. Korth; S. Sudarshan. FUNDAMENTOS DE BASES DE DATOS. Cuarta Edición. España: McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U, 2002.
- C. J. Date. Introducción A Los Sistemas De Bases De Datos. Séptima Edición. Méjico: Pearson Educación, 2001. ISBN: 968-444-419-2
- PostgreSQL Current Documentation. Disponible en: https://www.postgresql.org/docs

Diseño y Administración de Base de Datos

Unidad IV (bis)

Transacciones

Ing. Alejandro Mostovoi

CUVL - 2020

Índice

Gestión de transacciones **2**
Conceptos 2
Control de la concurrencia 5
Esquema de control de concurrencia multiversión (MVCC) 6
Implementación en PostgreSQL 8
Niveles de aislamiento 9
Read (un)Committed Isolation Level 12
Repeatable Read Isolation Level 13
Serializable Isolation Level 14
Utilización de transacciones 14

Lectura recomendada **15**

Bibliografía **16**

Gestión de transacciones

El término transacción hace referencia a un conjunto de operaciones que forman una única unidad lógica, es decir, que deben ser tratadas de forma atómica o en otras palabras como una única operación.

Resulta importante que, o bien se ejecuten completamente todas las acciones de una transacción, o bien, en caso de fallo, se deshagan los efectos parciales de la transacción. Esta propiedad se denomina **atomicidad**. Además, una vez ejecutada con éxito una transacción, sus efectos deben persistir en la base de datos: un fallo en el sistema no debe tener como consecuencia que la base de datos se olvide de una transacción que haya completado con éxito. Esta propiedad se denomina **durabilidad.**

En los sistemas de bases de datos en los que se ejecutan de manera concurrente varias transacciones, si no se controlan las actualizaciones de los datos compartidos existe la posibilidad de que las transacciones vean estados intermedios inconsistentes creados por las actualizaciones de otras transacciones. Esta situación puede dar lugar a actualizaciones erróneas de los datos almacenados en la base de datos. Por tanto, los sistemas de bases de datos deben proporcionar los mecanismos para aislar las transacciones de otras transacciones que se ejecuten de manera concurrente. Esta propiedad se denomina **aislamiento**.

Conceptos

Como se mencionó anteriormente, una transacción es un conjunto de instrucciones ejecutadas como una unidad, que accede y posiblemente actualiza varios datos.

Una transacción se inicia por la ejecución de un programa de usuario escrito en un lenguaje de manipulación de datos de alto nivel o en un lenguaje de programación (por ejemplo SQL, COBOL, C, C++ o Java), y está delimitado por instrucciones (o llamadas a función) de la forma ***inicio_transacción*** y ***fin_transacción***.

Con el objetivo de asegurar la integridad de los datos se necesita que el sistema de base de datos mantenga las siguientes propiedades:

- **Atomicidad**: o bien todas las operaciones de la transacción se realizan adecuadamente en la base de datos o ninguna de ellas.
- **Consistencia**: la ejecución aislada de la transacción (es decir, sin otra transacción que se ejecute concurrentemente) conserva la consistencia de la base de datos.

- **Aislamiento**: aunque se ejecuten varias transacciones concurrentemente, el sistema garantiza que para cada par de transacciones Ti y Tj, se cumple que para los efectos de Ti, o bien Tj ha terminado su ejecución antes de que comience Ti , o bien que Tj ha comenzado su ejecución después de que Ti termine. De este modo, cada transacción ignora al resto de las transacciones que se ejecuten concurrentemente en el sistema.
- **Durabilidad**: Tras la finalización con éxito de una transacción, los cambios realizados en la base de datos permanecen, incluso si hay fallos en el sistema.

Estas propiedades reciben el nombre de propiedades **ACID**; el acrónimo se obtiene de la primera letra de cada una de las cuatro propiedades en inglés (Atomicity, Consistency, Isolation and Durability, respectivamente).

Veamos un ejemplo para comprender mejor cómo impactan cada una de estas propiedades en la operatoria de una base de datos. Consideremos un sistema bancario (simplificado) donde se requiere actualizar los saldos de las cuentas de los clientes. Vamos a considerar, para simplificar el ejemplo, que los datos de la base de datos residen únicamente en disco y que las escrituras se realizan inmediatamente a disco.

Sea Ti una transacción para transferir $50 de la cuenta A a la cuenta B. Se puede definir dicha transacción como

```
Ti:
   leer (A);
   A := A - 50;
   escribir (A);
   leer (B);
   B := B + 50;
   escribir (B).
```

Veamos cómo aplican cada una de las propiedades **ACID**:

Consistencia: en este caso el requisito de consistencia es que la suma de A y B no sea alterada al ejecutar la transacción. Sin el requisito de consistencia, la transacción podría crear o destruir dinero. La responsabilidad de asegurar la consistencia de una transacción es del programador de la aplicación que codifica dicha transacción. La comprobación automática de las restricciones de integridad puede facilitar esta tarea.

Atomicidad: supóngase que justo antes de ejecutar la transacción Ti los valores de las cuentas A y B son de $1.000 y de $2.000, respectivamente. Supóngase ahora que durante la ejecución de la transacción Ti se produce un fallo que impide que dicha transacción finalice con éxito su ejecución. Ejemplos de este tipo de fallos pueden ser los fallos en la alimentación, los fallos del

hardware y los errores software. Además, supóngase que el fallo tiene lugar después de ejecutarse la operación escribir (A), pero antes de ejecutarse la operación escribir (B). Es ese caso, los valores de las cuentas A y B que se ven reflejados en la base de datos son $950 y $2.000. Se han perdido $50 de la cuenta A como resultado de este fallo.

En particular se puede ver que ya no se conserva la suma A + B. Así, como resultado del fallo, el estado del sistema deja de reflejar el estado real del mundo que se supone que modela la base de datos. Un estado así se denomina estado inconsistente.

Sin embargo, un sistema puede en algún momento alcanzar un estado inconsistente, incluso si la transacción Ti se ejecuta por completo, existe un punto intermedio en el que el valor de la cuenta A es de $950 y el de la cuenta B es de $2.000, lo cual constituye claramente un estado inconsistente. Este estado, sin embargo, se sustituye eventualmente por otro estado consistente en el que el valor de la cuenta A es de $950 y el de la cuenta B es de $2.050. De este modo, si la transacción no empieza nunca o se garantiza que se complete, un estado inconsistente así no será visible excepto durante la ejecución de la transacción. Ésta es la razón de que aparezca el requisito de atomicidad. Si se proporciona la propiedad de atomicidad, o todas las acciones de la transacción se ven reflejadas en la base de datos, o ninguna de ellas.

La idea básica que hay detrás de asegurar la atomicidad es la siguiente: el sistema de base de datos mantiene los valores antiguos (en disco) de aquellos datos sobre los que una transacción realiza una escritura y, si la transacción no completa su ejecución, los valores antiguos se recuperan para que parezca que la transacción no se ha ejecutado.

La responsabilidad de asegurar la atomicidad es del sistema de base de datos; en concreto, lo maneja un componente llamado componente de gestión de transacciones

Durabilidad: la propiedad de durabilidad asegura que, una vez que se completa con éxito una transacción, persisten todas las modificaciones realizadas en la base de datos, incluso si hay un fallo en el sistema después de completarse la ejecución de dicha transacción.

A partir de ahora se asume que un fallo en la computadora del sistema produce una pérdida de datos de la memoria principal, pero los datos almacenados en disco nunca se pierden. Se puede garantizar la durabilidad si se asegura que:

1. Las modificaciones realizadas por la transacción se guardan en disco antes de que finalice la transacción.
2. La información de las modificaciones realizadas por la transacción guardada en disco es suficiente para permitir a la base de datos reconstruir dichas modificaciones cuando el sistema se reinicie después del fallo

La responsabilidad de asegurar la durabilidad es de un componente del sistema de base de datos llamado componente de gestión de recuperaciones o gestor de integridad.

Aislamiento: incluso si se aseguran las propiedades de consistencia y de atomicidad para cada transacción, si varias transacciones se ejecutan concurrentemente, se pueden entrelazar sus operaciones de un modo no deseado, produciendo un estado inconsistente.

Por ejemplo, como se ha visto antes, la base de datos es inconsistente temporalmente durante la ejecución de la transacción para transferir fondos de la cuenta A a la cuenta B, con el total deducido escrito ya en A y el total incrementado todavía sin escribir en B. Si una segunda transacción que se ejecuta concurrente lee A y B en este punto intermedio y calcula A + B, observará un valor inconsistente. Además, si esta segunda transacción realiza después modificaciones en A y B basándose en los valores leídos, la base de datos puede permanecer en un estado inconsistente aunque ambas transacciones terminen.

Una solución para el problema de ejecutar transacciones concurrentemente es ejecutarlas secuencialmente, es decir, una tras otra. Sin embargo, la ejecución concurrente de transacciones produce notables beneficios en el rendimiento. Se han desarrollado otras soluciones que permiten la ejecución concurrente de varias transacciones. La propiedad de aislamiento asegura que el resultado obtenido al ejecutar concurrentemente las transacciones es un estado del sistema equivalente a uno obtenido al ejecutar una tras otra en algún orden.

La responsabilidad de asegurar la propiedad de aislamiento es de un componente del sistema de base de datos llamado componente de control de concurrencia.

Control de la concurrencia

Cuando se ejecutan concurrentemente varias transacciones, la base de datos puede dejar de ser consistente. Es necesario que el sistema controle la interacción entre las transacciones concurrentes, y dicho control se lleva a cabo mediante uno de los muchos mecanismos llamados esquemas de control de concurrencia.

Se pueden usar varios esquemas de control de concurrencia para asegurar la secuencialidad. Todos estos esquemas o bien retrasan (bloquean) una operación o bien abortan la transacción que ha realizado la operación. Los más comunes son los protocolos de bloqueo, los esquemas de ordenación por marcas temporales, las técnicas de validación y los esquemas multiversión.

Esquema de control de concurrencia multiversión (MVCC)

En los esquemas de control de concurrencia multiversión, donde Q es un dato, cada operación escribir(Q) crea una nueva versión de Q. Cuando se realiza una operación leer(Q) el gestor de control de concurrencia elige una de las versiones de Q que se va a leer.

El esquema de control de concurrencia debe asegurar que la elección de la versión que se va a leer se haga de tal manera que asegure la secuencialidad. Así mismo es crucial, por motivos de rendimiento, que una transacción sea capaz de determinar rápida y fácilmente la versión del elemento de datos que se va a leer.

Ordenación por marcas temporales multiversión

La técnica más frecuentemente utilizada en los esquemas multiversión es la de las marcas temporales. A cada transacción Ti del sistema se le asocia una única marca temporal estática que se denota MT(Ti). El sistema de bases de datos asigna dicha marca temporal antes de que la transacción comience su ejecución

A cada elemento de datos Q se el asocia una secuencia de versiones <Q 1, Q 2, ... , Qm>. Cada versión Qk contiene tres campos:

- contenido es el valor de la versión Qk.
- marca_temporal-E(Qk) es la marca temporal de la transacción que haya creado la versión Qk.
- marca_temporal-L(Qk) es la mayor marca temporal de todas las transacciones que hayan leído con éxito la versión Qk.

Una transacción —por ejemplo, Ti— crea una nueva versión Qk del elemento de datos Q realizando la operación escribir(Q). El campo contenido de la versión tiene el valor que ha escrito Ti. El sistema inicializa la marca_temporal-E y marca_temporal-L con MT(Ti). El valor marca_temporal-L se actualiza cada vez que una transacción Tj lee el contenido de Qk y marca_temporal-L(Qk) < MT(Tj).

El esquema de marcas temporales multiversión que se muestra a continuación asegura la secuencialidad. El esquema opera como sigue. Supóngase que la transacción T i realiza una operación leer(Q) o escribir(Q). Sea Qk la versión de Q cuya marca temporal de escritura es la mayor marca temporal menor o igual que MT(Ti).

1. Si la transacción T i ejecuta leer (Q), entonces el valor que se devuelve es el contenido de la versión Q k.

2. Si la transacción Ti ejecuta escribir(Q) y si MT(Ti) < marca_temporal-L(Qk), entonces la transacción Ti se retrocede. Caso contrario, si MT(Ti) = marca_temporal-E(Qk) se sobrescribe el contenido de Qk, y en otro caso se crea una nueva versión de Q.

La justificación de la regla 1 es que una transacción lea la versión más reciente/actual que viene antes de ella en el tiempo.

La segunda regla fuerza a que se aborte una transacción que realice una escritura «demasiado tarde». Con más precisión, si Ti intenta escribir una versión que alguna otra transacción haya leído, entonces no se puede permitir que dicha escritura tenga éxito.

El esquema de ordenación por marcas temporales multiversión tiene la propiedad deseable de que nunca falla una petición de lectura y nunca tiene que esperar. En sistemas de bases de datos típicos, en los cuales la lectura es una operación mucho más frecuente que la escritura, esta ventaja puede tener un mayor significado práctico.

Sin embargo, tiene dos propiedades no deseables. En primer lugar, la lectura de un elemento de datos necesita también la actualización del campo marca_temporal-L, lo que tiene como resultado dos accesos potenciales al disco en lugar de uno. En segundo lugar, los conflictos entre las transacciones se resuelven por medio de retrocesos en lugar de esperas. Esta alternativa puede ser costosa.

Bloqueo de dos fases multiversión

Este protocolo distingue entre transacciones de sólo lectura y transacciones de actualización. Las transacciones de actualización mantienen todos los bloqueos hasta el final de la transacción. Así, se puede secuenciar según su orden de terminación. Cada elemento de datos tiene una única marca temporal. La marca temporal no es en este caso una marca temporal basada en un reloj real, sino que es un contador, que se denomina contador_mt, que se incrementa durante el procesamiento del compromiso.

A las transacciones de sólo lectura se les asigna una marca temporal leyendo el valor actual de contador_mt antes de que comiencen su ejecución; para realizar las lecturas siguen el protocolo de ordenación por marcas temporales multiversión. Así, cuando una transacción de sólo lectura Ti ejecuta leer(Q), el valor que se devuelve es el contenido de la versión con la mayor marca temporal menor que MT(Ti).

Cuando una transacción de actualización lee un elemento obtiene un bloqueo compartido sobre él y lee la versión más reciente de dicho elemento de datos. Cuando una transacción de actualización quiere escribir un elemento, obtiene primero un bloqueo exclusivo sobre el elemento, y luego crea una nueva versión del elemento de datos. La escritura se realiza sobre la

nueva versión y la marca temporal de la nueva versión tiene inicialmente el valor ∞, el cual es mayor que el de cualquier marca temporal posible.

Una vez que la transacción Ti ha completado sus acciones realiza el procesamiento del compromiso como sigue:

1. en primer lugar, Ti asigna la marca temporal de todas las versiones que ha creado al valor de contador_mt más 1;
2. después Ti incrementa contador_mt en 1. Sólo se permite que realice el procesamiento del compromiso a una transacción de actualización a la vez.

Como resultado, las transacciones que comiencen después de que Ti incremente el contador_mt se observarán los valores que Ti ha actualizado, mientras que aquellas que comiencen antes de que Ti incremente el contador_mt se observarán los valores anteriores a la actualización de Ti. En ambos casos las transacciones de sólo lectura no tienen que esperar nunca por los bloqueos.

El bloqueo de dos fases multiversión o variaciones de éste se usan en algunos sistemas de bases de datos comerciales.

Implementación en PostgreSQL

PostgreSQL mantiene la coherencia de los datos mediante el uso de un modelo de múltiples versiones (Control de concurrencia de múltiples versiones, MVCC). Esto significa que cada instrucción SQL ve una instantánea de los datos (una versión de la base de datos), independientemente del estado actual de los datos subyacentes. Esto evita que se vean datos inconsistentes producidos por transacciones concurrentes que realizan actualizaciones en las mismas filas de datos, proporcionando aislamiento de transacciones para cada sesión de base de datos. MVCC, al evitar las metodologías de bloqueo de los sistemas de base de datos tradicionales, minimiza la contención de bloqueo para permitir un rendimiento razonable en entornos multiusuario.

La principal ventaja de usar el modelo MVCC de control de concurrencia en lugar de bloquear es que en MVCC los bloqueos adquiridos para consultar (leer) datos no entran en conflicto con los bloqueos adquiridos para escribir datos, por lo que la lectura nunca bloquea la escritura y la escritura nunca bloquea la lectura. PostgreSQL mantiene esta garantía incluso cuando proporciona el nivel más estricto de aislamiento de transacciones mediante el uso de un innovador nivel de aislamiento de instantáneas serializables (SSI).

Las funciones de bloqueo a nivel de tabla y de fila también están disponibles en PostgreSQL para aplicaciones que generalmente no necesitan un aislamiento total de transacciones y prefieren administrar explícitamente puntos particulares de conflicto.

Niveles de aislamiento

El estándar SQL define cuatro niveles de aislamiento de transacciones:

- Read uncommitted
- Read committed
- Repeatable read
- Serializable

El más estricto es ***Serializable***, el mismo establece que cualquier ejecución concurrente de un conjunto de transacciones serializables está garantizado para producir el mismo efecto que ejecutarlas una a la vez en algún orden.

En el estudio de las transacciones concurrentes, surgen algunos fenómenos que deben ser tenidos en cuenta al momento de diseñar una aplicación. Estos fenómenos son problemas generados por la posible concurrencia de las transacciones. Tengamos en cuenta que una base de datos puede ser visto como un un recurso compartido por varios procesos ya sea que se ejecutan en la misma máquina o en máquinas distribuídas a través de la red. Cabe destacar que estos fenómenos no se producen cuando se utiliza el nivel de aislamiento ***Serializable.*** Los eventos son:

- **Lectura no confirmada o sucia (dirty read):** una transacción lee datos escritos por otra transacción concurrente que no confirmó los datos, es decir, la transacción está trabajando con datos temporales.
  Por ejemplo:

Transacción A	Transacción B
`BEGIN TRANSACCTION` `  update products set uprice = uprice * 2;`	

...	`SET TRANSACTION ISOLATION LEVEL READ UNCOMMITED;`
	
	`DECLARE vuprice float;`
	
	`select uprice into vuprice from products where product_id = 24;`

`rollback transaction;`	...
	`insert into fac_detalle (...., uprice, ....) values (..., vuprice, ....)`

En este ejemplo, la transacción B utiliza un precio unitario erróneo ya que la transacción A deshizo los cambios.

- **Análisis inconsistente o Lectura no repetible (non repeatable read):** una transacción vuelve a leer los datos que ha leído anteriormente y encuentra que los datos han sido modificados por otra transacción (que se confirmó desde la lectura inicial).

  Por ejemplo: si se ejecuta un proceso de larga duración, digamos un reporte mensual de facturación, puede ocurrir que durante la ejecución del reporte, se actualicen algunas ventas (otras transacciones se ejecuten) lo que llevará a generar datos erróneos por ejemplo que el total no coincida con el detalle.

Transacción A (generación reporte)	Transacción B (pueden ser varias)

`select sum((pu * cant) * (1-descuento))`
`into total`
`from fac_detalles;`	

	`update fac_detalles set descuento`
	`= 0.4`
	`where prod_id = 10;`

`select fac_id, pu, cant, descuento` `from fac_destalles;`	

  En este caso, el total será diferente del detalle.

- **Lectura fantasma (phantom read):** similar a la situación anterior, la lectura fantasma es una anomalía en la cual una transacción vuelve a ejecutar una consulta que devuelve un conjunto de filas que satisfacen una condición de búsqueda y encuentra que el conjunto de filas que satisfacen la condición ha cambiado debido a otra transacción confirmada recientemente.
  Por ejemplo: si se ejecuta un proceso de larga duración, digamos un reporte mensual de facturación, puede ocurrir que durante la ejecución del reporte, se realicen otras ventas (otras transacciones se ejecuten) lo que llevará a generar datos erróneos por ejemplo que el total no coincida con el detalle.

Transacción A (generación reporte)	Transacción B (pueden ser varias)

`select sum((pu * cant) * (1-descuento))`
`into total`
`from fac_detalles;`	

	`insert into fac_detalles (....) values (...);`
`select fac_id, pu, cant, descuento from fac_destalles;`	

- **Anomalía de serialización (serialization anomaly):** se puede producir una situación de modificaciones perdidas cada vez que 2 conexiones modifican los mismos datos una después de la otra, en otras palabras, una condición de carrera.

Transacción A (generación reporte)	Transacción B (pueden ser varias)
`Estado inicial: pu = 1`	
`select pu into vpu from products where p_id = 23;`	
	`select pu into vpu from products where p_id = 23;`
`update products set pu = vpu * 1.2 where p_id = 23;`	
	`update products set pu = vpu * 1.4 where p_id = 23;`
`select vpu * 1.2, pu from products where p_id = 23;`	`select pu from products where p_id = 23;`
`vpu | pu | ---------- 1.2 | 1.4`	` pu | ------ 1.4 |`

Este problema se puede prevenir escribiendo instrucciones atómicas, por ejemplo:

```
update products set pu = pu * 1.2 where p_id = 23;
```

En PostgreSQL se pueden utilizar cualquiera de los 4 niveles de aislamiento mencionados, aunque sólo 3 se encuentren implementados.

La siguiente tabla muestra los niveles de aislamiento y los fenómenos que pueden producirse en cada uno.

Nivel/Fenómeno	Dirty read	Nonrepeatable read	Phantom read	Serialization anomaly
Read uncommited	Permitido, pero no en Postgres	✅	✅	✅
Read commited	❌	✅	✅	✅
Repeatable read	❌	❌	Permitido, pero no en Postgres	✅
Serializable	❌	❌	❌	❌

Para establecer el nivel de aislamiento de una transacción, se utiliza el comando SET TRANSACTION (https://www.postgresql.org/docs/current/sql-set-transaction.html).

Read (un)Committed Isolation Level

Es el nivel de aislamiento predeterminado en PostgreSQL. Cuando una transacción usa este nivel de aislamiento, una consulta SELECT solo ve los datos confirmados antes de que comenzara la consulta; nunca ve datos no confirmados o cambios cometidos durante la ejecución de la consulta por transacciones concurrentes.

En efecto, una consulta SELECT ve una instantánea (snapshot) de la base de datos en el momento en que la consulta comienza a ejecutarse. Esto significa que si alguna transacción concurrente está modificando los datos, la consulta SELECT no verá la modificación hasta que los cambios mencionados se hayan confirmado. Sin embargo, sí ve los efectos de las actualizaciones anteriores ejecutadas dentro de su propia transacción, aunque aún no se hayan confirmado.

Por otra parte, operaciones como INSERT, UPDATE o DELETE su comportamiento es un poco distinto en el sentido que sólo verán los datos confirmados antes de comenzar la operación. Ahora bien, puede ocurrir que al momento de comenzar la operación, los datos determinados por la cláusula WHERE estén siendo modificados por otra transacción concurrente. En ese caso, el UPDATE o DELETE se pondrá en espera a que la transacción concurrente confirme o deshaga

los cambios. Si la transacción concurrente no aplica los cambios, el UPDATE o DELETE procede como si nada hubiese ocurrido. Si la transacción concurrente confirma los cambios, entonces el WHERE vuelve a evaluarse para definir si se debe o no seleccionar la fila en dicha operación.

```
BEGIN;
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 12345;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 7534;
COMMIT;
```

Si 2 transacciones como la mostrada se ejecutan de forma concurrente, no se producen errores de consistencia por los motivos ya explicados. La segunda transacción deberá esperar a que la primera finalice para luego actualizar los datos.

Veamos otro ejemplo: supongamos que la situación actual es que existen 2 registros en la tabla *website* donde uno posee 9 hits y el otro 10.

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- run from another session:  DELETE FROM website WHERE hits = 10;
COMMIT;
```

Analicemos qué ocurre en esta situación:

1. El where del DELETE pre-seleccionará 1 fila, la que posee 10 en el campo hits
2. Se ejecuta el update que actualiza el campo hits en todos los registros, llevando el 9 a 10 y el 10 a 11.
3. Cuando finaliza el UPDATE, y el DELETE obtiene el bloqueo sobre las filas, las filas preseleccionadas ya no cumplen la condición por lo que no eliminará ninguna fila.

Como resumen podemos decir que este nivel de aislamiento si bien impide lecturas no confirmadas, permite que otras conexiones actualicen datos previamente leídos (lecturas no repetibles) o inserten nuevas filas (lecturas fantasma).

Tener en cuenta que otros motores de base de datos implementan por separado los niveles aquí presentados.

Para la mayoría de las aplicaciones este nivel de aislamiento proporciona un buen balance entre bloqueos y rendimiento.

Repeatable Read Isolation Level

El nivel de aislamiento de lectura repetible sólo ve los datos confirmados antes de que comenzara la transacción; nunca ve datos no confirmados o cambios cometidos durante la ejecución de transacciones por transacciones concurrentes. (Sin embargo, la consulta ve los

efectos de las actualizaciones anteriores ejecutadas dentro de su propia transacción, aunque aún no se hayan confirmado). Esta es una garantía más sólida que la requerida por el estándar SQL para este nivel de aislamiento y evita todos los fenómenos. descrito previamente excepto por las anomalías de serialización.

Este nivel es diferente de Lectura confirmada dado que una consulta en una transacción de lectura repetible ve siempre los mismos datos sin que las actualizaciones realizadas en el resto de las transacciones los afecten.

Por lo tanto, los comandos SELECT sucesivos dentro de una sola transacción ven los mismos datos, es decir, no ven los cambios realizados por otras transacciones que confirmaron los datos después de que comenzó la transacción.

Tener en cuenta que este nivel de aislamiento en otros motores de base de datos, si bien asegura lecturas consistentes no impide la aparición de lecturas fantasmas. En Postgres sí las impide.

Serializable Isolation Level

Este nivel de aislamiento es el nivel más estricto emulando las transacciones serializables como si se ejecutaran una detrás de la otra.

Utilización de transacciones

Para utilizar transacciones dentro de nuestras consultas se utilizan los comandos

- BEGIN TRANSACTION: para comenzar un bloque de operaciones
- COMMIT: para confirmar las operaciones del bloque.
- ROLLBACK: para deshacer las operaciones del bloque.

Ejemplo I:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00 WHERE name = 'Alice';
COMMIT;
```

Ejemplo II:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00 WHERE name = 'Alice';
SELECT balance into vbalance from accounts where name = ‘Alice’;
IF vbalance < 0 THEN
    ROLLBACK;
ELSE
    COMMIT;
END IF;
```

Lectura recomendada

- https://www.postgresql.org/docs/10/tutorial-transactions.html
- https://www.postgresql.org/docs/current/mvcc.html

Bibliografía

- Gavin Powell. Beginning Database Design. Indianapolis: Wiley Publishing, Inc, 2006. ISBN-13: 978-0-7645-7490-0.
- Abraham Silberschatz; Henry F. Korth; S. Sudarshan. FUNDAMENTOS DE BASES DE DATOS. Cuarta Edición. España: McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U, 2002.
- C. J. Date. Introducción A Los Sistemas De Bases De Datos. Séptima Edición. Méjico: Pearson Educación, 2001. ISBN: 968-444-419-2
- PostgreSQL Current Documentation. Disponible en: https://www.postgresql.org/docs

Diseño y Administración de Base de Datos

Unidad V

Normalización

Ing. Alejandro Mostovoi

Año - 2023

Revisión: 1.1

Tabla de versiones

Autor	Fecha	Versión	Descripción
Alejandro Mostovoi	01/10/2018	1.0	Versión inicial
Alejandro Mostovoi	04/10/2023	1.1	Mejoras en redacción y aclaraciones

Índice

Introducción **2**
- Anomalías 3
  - Insert 3
  - Delete 3
  - Update 3

Formas Normales **3**
- Primera Forma Normal (1FN) 4
- Segunda Forma Normal (2FN) 4
- Tercera Forma Normal (3FN) 6
- Cuarta, Quinta y Sexta Forma Normal 7
- Boyce-Codd (BCNF) 8
- Cuarta Forma Normal (4FN) 10
- Quinta Forma Normal (5FN) 10
- Sexta Forma Normal (DKNF) 12

Bibliografía **12**

Introducción

Definición	La normalización es una secuencia de pasos mediante los cuales se crea y mejora un modelo de base de datos relacional.

Los pasos involucrados en el proceso de normalización se los denomina Formas Normales. Estas formas normales, aplicadas secuencialmente son las que permiten la creación y la mejora de un modelo de base de datos.

Como dijimos, la normalización es un proceso que involucra una serie de 6 pasos que van desde la primera forma normal a la sexta forma normal. Por lo general sólo se implementan los primeros 3 pasos, es muy raro encontrar modelos que implementen alguno de los otros niveles.

El objetivo principal de la normalización es eliminar la duplicación de datos y así minimizar la información redundante. El resultado de este proceso, se puede decir que, es una mejor organización de los datos y un mejor aprovechamiento del espacio físico.

Anomalías

En las bases de datos relacionales se trata de eliminar las anomalías. Las anomalías son un efecto no deseado desde el punto de vista de la gestión de los datos ya que pueden conducir a que los mismos se corrompan, provocando a futuro pérdida de información o lo que es peor, información errónea. Existen varios tipos de anomalías:

Insert

Producida cuando se inserta un registro a una tabla hija sin que exista el registro relacionado en la tabla padre.

Delete

Producida cuando se elimina un registro de una tabla padre dejando huérfanos uno o varios registros en tablas hijas.

Update

Al igual que en el caso de *Delete*, tanto la tabla padre como las tablas hijas deben ser consideradas durante la operación para evitar la aparición de registros huérfanos.

Formas Normales

Las formas normales son aplicadas a las tablas de una base de datos. Decir que una base de datos está en la forma normal N es decir que todas sus tablas están en la forma normal N.

Como mencionamos anteriormente las bases de datos relacionales se normalizan para:

- Evitar la redundancia de los datos.
- Eliminar inconsistencias generadas al actualizar y/o eliminar datos de las tablas.
- Proteger la integridad de los datos.

Primera Forma Normal (1FN)

Principalmente consiste en eliminar los grupos de campos repetidos, creando nuevas tablas para dichos campos, estableciendo una relación 1 - N entre la tabla original y las nuevas.

Los pasos para lograr la 1FN son:

- Eliminar los grupos de campos repetidos
- Definir claves primarias
- Todos los registros de todas las tablas deben ser identificables por la clave primaria.
- Todos los campos, excepto los de la clave primaria, deben depender de la clave primaria.
- Crear nuevas tablas para mover los grupos de campos repetidos de la tabla original.

Ejemplo:

a.

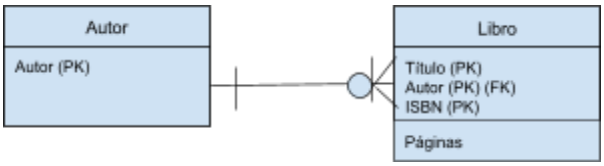
Autor	Título_01	Páginas_01	Título_02	Páginas_02	ISBN_01
Tolkien	El señor de los anillos	1000	El Hobbit	300	1

b.

Autor	Títulos	Páginas	ISBNs
Tolkien	El señor de los anillos, El Hobbit	1000, 300	1, 2

Tanto en el ejemplo ‘a’ como ‘b’, vemos que existen grupos de campos para el *título* y la *cantidad de páginas.*

Para llevar esta tabla, que se encuentra en 0FN, a 1FN debemos eliminar los grupos de campos y buscar y generar las claves primarias correspondientes. Es probable (casi una certeza) que surjan nuevas tablas en este paso



Si en cualquiera de los ejemplos mencionados previamente existieran, por ejemplo, datos de la "Editorial" del libro, esos datos deberían ser extraídos en una nueva tabla ya que los mismos no son datos que dependan de la clave primaria del libro.

Segunda Forma Normal (2FN)

Como precondición, la tabla debe estar en 1FN.

Mientras que en el paso anterior, se eliminaban los campos repetidos, en este paso, el objetivo principal es eliminar los valores estáticos (repetidos) y pasarlos a nuevas tablas estableciendo relaciones N - 1 entre la tabla nueva y la tabla original.

Desde el punto de vista matemático de la relación, una relación 1-N o N-1 son exactamente iguales lo que cambia en este paso es el enfoque y el objetivo final que es eliminar los datos duplicados.

Otro de los objetivos logrados es que las nuevas tablas, obtienen una clave primaria conformada por un único campo.

Los pasos para lograr la 2FN son:

- La tabla debe estar en 1FN
- Todos los campos que no son parte de la clave primaria deben poder ser determinados a partir de la clave primaria (dependencia funcional)
- Eliminar las dependencias parciales. Una dependencia parcial se da cuando un campo depende parcialmente de la clave principal.
- Crear nuevas tablas para mover los campos que dependen parcialmente de la clave principal.

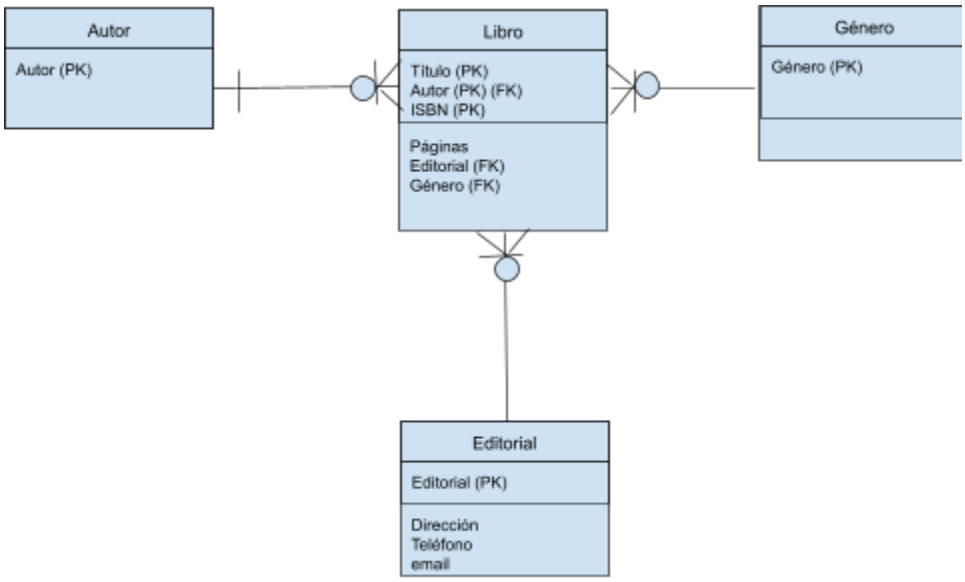
Ejemplo:

Supongamos que en las tablas del modelo de libros tenemos además datos sobre la editorial y sobre el género del libro

Autor	Título	ISBN	Páginas	Editorial	Dirección	Género
Tolkien	El señor de los anillos	1	1000	Cosme Fulanito	Calle Siempre Viva 1000	Ciencia Ficción

En esta tabla vemos que tanto los datos del Editorial (campos Editorial y Dirección) como el Género, no son datos que tengan una dependencia directa con la clave primaria, estos son los campos candidatos a remover y trasladar a nuevas tablas.

Tercera Forma Normal (3FN)



Como precondición, la tabla debe estar en 2FN.

En este paso se trata de eliminar las dependencias transitivas, que se dan cuando un campo tiene una dependencia funcional con otro campo que no forma parte de la clave primaria.

Básicamente todos los campos que no formen parte de la clave primaria deben depender directamente de la clave primaria.

Casos

Relaciones N - M. Para lograr que las tablas pasen a estar en 3FN, hay que generar una tabla intermedia que permita obtener registros y atributos particulares sin repetición.

Otro caso es cuando se tienen datos en común entre 2 o más tablas. Por lo general vamos a encontrar una dependencia transitiva y esta debe ser eliminada. Los campos comunes deben ser movidos a una nueva tabla.

Como se mencionó en la introducción, las dependencias transitivas deben ser eliminadas moviendo los campos involucrados hacia una nueva tabla.

Por último si existen campos calculados, los mismos deberían[1] ser eliminados de las tablas.

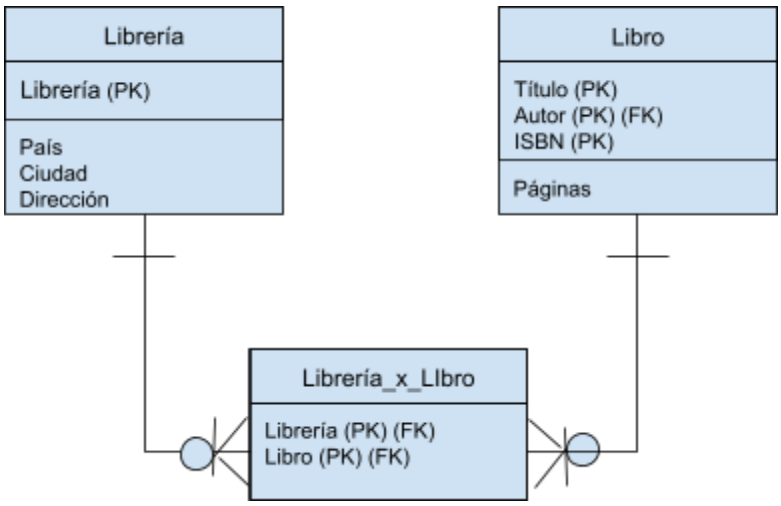
Ejemplo

Supongamos que a nuestro modelo se agregan las librerías, y necesitamos saber qué libros están en cada librería, podríamos tener en primera instancia:

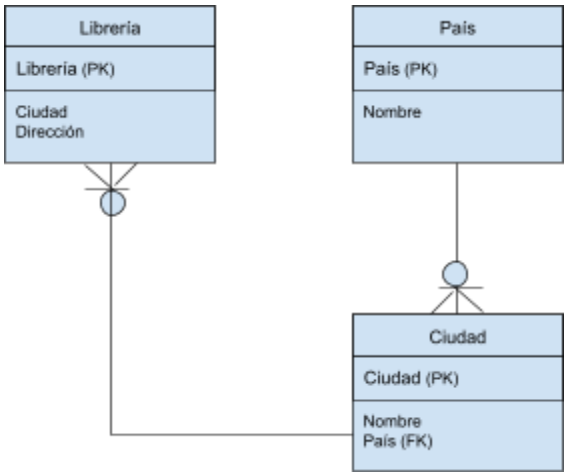
Librería	País	Ciudad	Dirección	Libro
La nueva	Argentina	CABA	xxx 123	Object Oriented Análysis and Design
La nueva 2	Argentina	CABA	xxx 123	Clean Code
El rincón del libro	Argentina	CABA	yyy 123	Object Oriented Análysis and Design
El rincón del libro	Argentina	CABA	yyy 123	Clean Code

Claramente tenemos una relación N-M y esta tabla no se encuentra en 3FN por motivos que ya mencionamos anteriormente. La solución será:

[1] Muchas veces, por cuestiones de rendimiento no está mal tener campos calculados en las tablas.



Como vemos este modelo cuenta con otros puntos a atacar como son las dependencias parciales y las dependencias transitivas



En este último modelo eliminamos las dependencias parciales de la librería con el país y la ciudad y además la dependencia transitiva entre la ciudad y el país.

Cuarta, Quinta y Sexta Forma Normal

Además de las descritas en este apunte, existen 4 formas normales más que son

Boyce-Codd (BCNF)

Como precondición, la tabla debe estar en 3FN.

El objetivo de este paso es el de separar todas las claves candidatas en nuevas tablas. Una clave candidata es un campo o grupo de campos que podrían ser la clave primaria de la tabla.

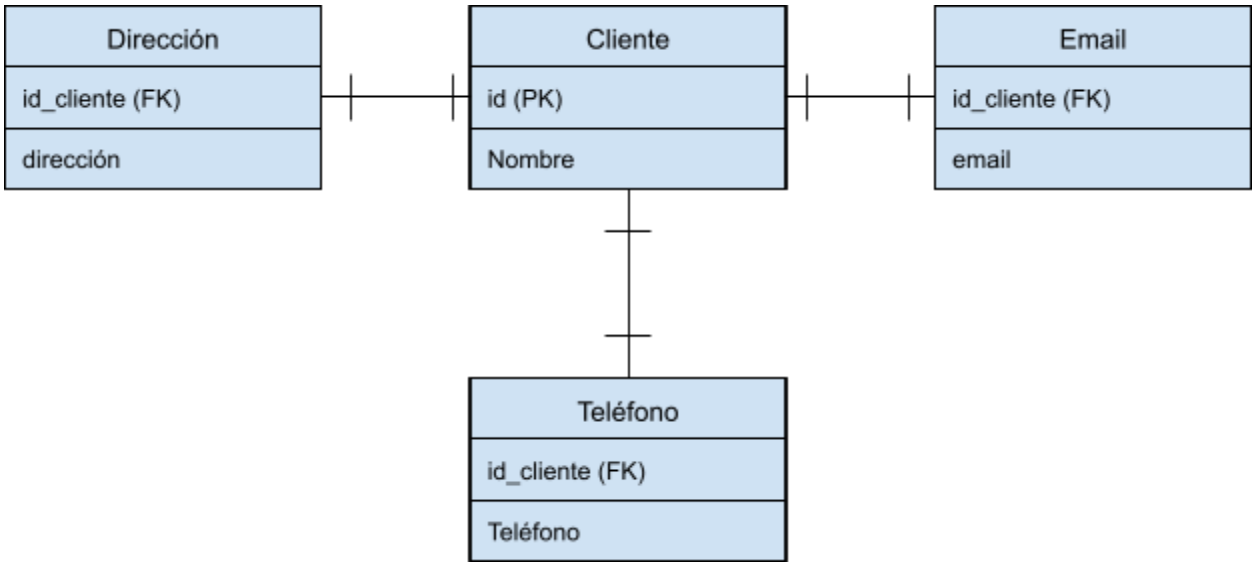
Este es un caso especial de la 3FN, ya que si sólo existe 1 clave candidata la 3FN y BCNF son iguales.

En caso de que existan varias claves candidatas, deben ser llevadas a nuevas tablas. El problema de este enfoque es que se degrada el rendimiento de las consultas.

Ejemplo

Cliente
id (PK)
Nombre Dirección Teléfono Email

Como la dirección, si contemplamos el código postal; el teléfono, si contemplamos el código de país y área; y el email son campos que poseen valores únicos, cualquiera de esos tres datos serían posibles claves primarias en sí mismas, por lo tanto la relación *Cliente* para estar en BCNF debe convertirse en 3 relaciones 1 - 1 de la siguiente manera:



Como ventajas de este enfoque podemos decir que permite eliminar campos con valores nulos (valor null). Por ejemplo, si un cliente no posee *teléfono* en la tabla original, ese campo quedará

vacío, es decir, con valor null, mientras que en el modelo BCNF no existiría el registro en la tabla *Teléfono*.

Pero también hay que tener en cuenta el rendimiento de las consultas ya que en el modelo original, para obtener todos los datos de 1 cliente sólo hay que acceder a una tabla, mientras que en el segundo hay que acceder a 4 tablas.

Cuarta Forma Normal (4FN)

Como precondición, la tabla debe estar en 3FN o BCNF.

Transforma los campos que poseen dependencias de valores múltiples en dependencias funcionales de forma tal que todos los campos tengan una dependencia funcional con la clave primaria de la tabla.

Este caso se da cuando en una tabla tenemos campos que poseen múltiples valores (separados por coma, por ejemplo) que no están relacionados uno con el otro. Estos deben ser llevados a nuevas tablas y relacionados con la tabla original.

Por lo general esto dará origen a múltiples relaciones 1 - N o N - M

Ejemplo

Empleados
Legajo (PK)
Nombre Habilidades Certificaciones

Legajo	Nombre	Habilidades	Certificaciones
1	Juan Perez	Analista, Programador	Zend PHP, C++
2	Ana Martinez	Project Manager, Tester, Analista	PMI, Testing X

En este caso hay que transformar la tabla *Empleados*, generando 2 relaciones N-M una para las Habilidades y otra para las Certificaciones. Obviamente habrá que crear las tablas intermedias que relacionen un empleado con sus habilidades y otra con sus certificaciones.

Básicamente, 4FN separa conjuntos de datos en registros, haciendo que cada registro individual sea más fácil de acceder cuando se realiza una búsqueda de coincidencia exacta.

Quinta Forma Normal (5FN)

Como precondición, la tabla debe estar en 4FN.

La 5FN también es conocida como *Quinta Forma Normal de Proyección-Unión*.

En este paso se trata de eliminar las dependencias cíclicas que pueda tener una tabla y de reducir la redundancia de los datos.

Una dependencia cíclica en el contexto de bases de datos sucede cuando la clave primaria de una tabla está compuesta por 3 o más campos y esos campos están relacionados de a pares entre sí.

El término *Proyección* se debe a que la tabla original se separa en varias tablas que contienen subconjuntos de datos (proyección) y el término *Unión* a que al realizar una consulta de todas las tablas se debe obtener el mismo resultado que se exhibía en la tabla original (join).

Ejemplo

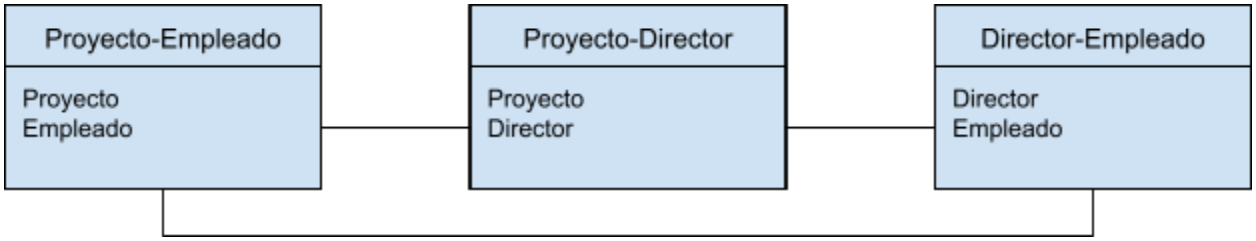
Empleados
Proyecto (PK) Empleado (PK) Director (PK)
...

Proyecto	Empleado	Director
Análisis	X	A
Análisis	Y	B
Análisis	Z	B
Diseño	W	C
Diseño	X	C

Algunos de los problemas que podemos ver en este caso son la redundancia de datos, el hecho de que al insertar un nuevo dato habrá que hacerlo tantas veces como combinaciones se necesiten.

La 5FN nos propone generar tablas de proyección para eliminar esta redundancia de datos. La forma de hacerlo es generar 1 tabla por cada par de campos relacionados, en este caso tendremos 3 tablas:

- Proyecto-Empleado
- Proyecto-Director
- Empleado-Director



Sobre el DER del ejemplo, las líneas que relacionan las tablas no son FK, sino que simplemente están expresando una semi-relación entre las mismas.

En la práctica ésta FN no es muy utilizada debido a que se generan muchas tablas y se puede degradar el rendimiento de las consultas

Sexta Forma Normal (DKNF)

Domain Key Normal Form (DKNF) es a menudo un estado conceptual de un modelo de base de datos relacional en oposición a un proceso de transformación final. DKNF es la última forma normal y esencialmente describe cómo debería aparecer un modelo de base de datos completamente normalizado:

- No puede haber anomalías de inserción, cambio o eliminación de datos. En otras palabras, cada registro en la base de datos debe ser accesible directamente de todas las maneras, de modo que no se produzcan errores.
- Cada registro en cada tabla debe ser identificable de manera única y estar directamente relacionado con la clave principal en su tabla. Esto significa que cada campo en cada tabla está directamente determinado por la clave primaria en su tabla.
- Toda validación de datos se realiza dentro del modelo de base de datos. En lo que respecta al rendimiento de la aplicación y de la base de datos, esto es casi siempre un estado extremadamente indeseable en un entorno comercial. Es mejor dividir la

funcionalidad entre la base de datos y las aplicaciones. Algunos pueden llamar a esto reglas comerciales. Por lo general, es de conocimiento común que algunas implementaciones de reglas comerciales a menudo se dividen de manera más efectiva entre la base de datos y las aplicaciones.

Bibliografía

- Gavin Powell. Beginning Database Design. Indianapolis: Wiley Publishing, Inc, 2006. ISBN-13: 978-0-7645-7490-0.
- Abraham Silberschatz; Henry F. Korth; S. Sudarshan. FUNDAMENTOS DE BASES DE DATOS. Cuarta Edición. España: McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U, 2002.
- C. J. Date. Introducción A Los Sistemas De Bases De Datos. Séptima Edición. Méjico: Pearson Educación, 2001. ISBN: 968-444-419-2

Diseño y Administración de Base de Datos

Unidad VI

Funciones y Triggers

Ing. Alejandro Mostovoi

CUVL - 2021

Índice

Funciones **2**
- Sintaxis 2
- Tipos de funciones 3
  - Veamos algunos ejemplos 3
- Trabajando con excepciones 5

Triggers **7**
- Tipos y Usos 7
- Registros especiales 8
- Creación de triggers 8

Lectura recomendada **10**

Bibliografía **11**

Funciones

Las funciones de PostgreSQL, también conocidas como Procedimientos almacenados (Store Procedures), como en todos los lenguajes de programación, tienen por objetivo encapsular cierta lógica de forma tal que podamos obtener algún resultado a partir de su invocación. Al igual que en un lenguaje de programación procedural, éstas funciones pueden recibir parámetros y devolver un único valor como resultado. Respecto de esto último, cabe mencionar que los valores o bien pueden ser de tipos de datos nativos como enteros o cadenas o bien pueden ser de tipo registro; este último permite que una función devuelva, tal como mencionamos en otras unidades, una tabla.

Las funciones permiten también la reutilización de la base de datos ya que otras aplicaciones pueden interactuar directamente con ellas en lugar de hacerlo a través de código u otras consultas, garantizando de esta forma un comportamiento homogéneo.

PostgreSQL tiene la particularidad de que las funciones se pueden desarrollar utilizando lenguaje SQL, PL / pgSQL, C, Python, etc.

Sintaxis

```
CREATE OR REPLACE FUNCTION <nombre>([[<mode>] <arg1> <type>, ...,[<mode>]
<argN> <type>]) RETURNS <ret_type> AS $$
DECLARE <var_name> <var_type>
BEGIN
...;
END;
$$ LANGUAGE <language>;
```

mode: IN|OUT|INOUT

type: tipo de cada argumento. Si el tipo está basado en alguna columna, se puede utilizar la siguiente expresión: <table_name>.<column_name>%TYPE

ret_type: tipo de retorno de la función. Puede ser un tipo básico o TABLE, en este caso hay que especificar las columnas y sus tipos.

language: puede ser sql; plpgsql; plpythonu

Tipos de funciones

Los tipos de funciones están determinados principalmente por el lenguaje que se utilice para programarlas.

En el caso de ***SQL*** sólo podrán efectuarse operaciones especificadas en dicho lenguaje. Si requerimos especificar una lógica más compleja, podemos utilizar ***plpgsql*** que nos habilita el uso de estructuras de decisión, bucles, etc.

También existe el tipo ***plpythonu*** que nos permite escribir las funciones en python, con todo su potencial.

Veamos algunos ejemplos

Utilizando lenguaje sql:

```
-- Ejemplo de función que permite actualizar un registro de una tabla
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS integer AS $$
  UPDATE bank
    SET balance = balance - debit
    WHERE accountno = tf1.accountno;
  SELECT 1;
$$ LANGUAGE SQL;


-- Ejemplo de función cuyo tipo de retorno es una tabla con 2 campos
CREATE FUNCTION dup(int) RETURNS TABLE(f1 int, f2 text) AS $$
  SELECT $1, CAST($1 AS text) || ' is text'
$$ LANGUAGE SQL;


SELECT * FROM dup(42);

-- Genera la siguiente salida
 f1 |     f2
----+------------
 42 | 42 is text
(1 row)
```

Cabe destacar que el segundo ejemplo, la función dup podría ser utilizada en la cláusula from de alguna otra consulta ya que devuelve una tabla.

Utilizando lenguaje plpgsql:

En caso de necesitar implementar cierta lógica que involucre toma de decisiones, iteraciones, etc, debemos desarrollar la función utilizando lenguaje plpgsql.

```
-- Ejemplo de función que valida un par de credenciales
CREATE OR REPLACE FUNCTION check_password(uname TEXT, pass TEXT) RETURNS BOOLEAN AS
$$
DECLARE passed BOOLEAN;
BEGIN
  passed := false;
  SELECT (password = $2) INTO passed
  FROM cliente
  WHERE name = $1;

  -- en caso de que el query no retorne filas, la variable passed contiene NULL.
  IF passed IS NULL THEN
       passed := false;
  END IF;

  RETURN passed;
END;
$$ LANGUAGE plpgsql;


-- Ejemplo de función que suma 2 enteros si y sólo si ambos son positivos
CREATE FUNCTION sum_positive(n1 integer, n2 integer) RETURNS integer AS $$
DECLARE sum integer;
BEGIN
  IF(n1 > 0 AND n2 > 0) THEN
    RETURN n1+n2;
  ELSE
    RAISE NOTICE ‘Error!!’;
  END IF;
END;
$$ LANGUAGE plpgsql;


-- Ejemplo de función que devuelve el registro con el id mayor.
CREATE OR REPLACE FUNCTION last_record() RETURNS provincias AS $$
DECLARE  t2_row provincias%ROWTYPE;
BEGIN
       select * into t2_row
       from provincias order by prv_id desc limit 1;

       return t2_row;
END;

$$ LANGUAGE plpgsql;
```

Es interesante analizar el segundo ejemplo, la función *sum_positive*, ya que como se puede ver utiliza la instrucción `RAISE` para lanzar una excepción en caso de que alguna validación no se cumpla o se quiera/necesite informar de algún error producido dentro de la función.

Para más información sobre el lenguaje plpgsql ver:

- https://www.postgresql.org/docs/current/static/plpgsql.html

Trabajando con excepciones

Como vimos en la sección anterior, si desde una función debemos lanzar una excepción, podemos utilizar la instrucción *RAISE* para realizarlo.

La sintaxis de la instrucción RAISE es la siguiente:

```
RAISE [nivel] [mensaje]
```

nivel: permite indicar el nivel de error. Puede ser uno de DEBUG, LOG, INFO, NOTICE, WARNING, o EXCEPTION. El nivel EXCEPTION es la opción por defecto y es el único de todos los niveles que permite cortar la ejecución de la función.

mensaje: permite generar un mensaje para describir el error.

Cabe destacar que en el archivo *postgresql.conf* podemos editar el nivel de los mensajes que deseamos registrar. Se puede configurar las variables *log_min_messages* y *client_min_messages* para configurar el nivel de los mensajes a reportar en el archivo de log y al cliente respectivamente.

Retomando el ejemplo anterior, tenemos:

```
-- Ejemplo de función que valida un par de credenciales
CREATE OR REPLACE FUNCTION check_password(uname TEXT, pass TEXT) RETURNS BOOLEAN AS
$$
DECLARE passed BOOLEAN;
BEGIN
  passed := false;
  SELECT (password = $2) INTO passed
  FROM cliente
  WHERE name = $1;

  -- en caso de que el query no retorne filas, la variable passed contiene NULL.
  IF passed IS NULL THEN
```

```
    RAISE LOG ‘El usuario % no existe’, uname;
       passed := false;
  END IF;

  RETURN passed;
END;
$$ LANGUAGE plpgsql;


-- Al ejecutar la función
select check_password('juan', 'clave_1234');

-- Obtenemos la salida
 check_password
----------------
 f
(1 row)


-- Y si vemos el archivo de log, veremos el siguiente mensaje
2021-10-28 03:45:13.093 UTC [49] LOG:  El usuario juan no existe
2021-10-28 03:45:13.093 UTC [49] CONTEXT:  PL/pgSQL function
check_password(text,text) line 10 at RAISE
2021-10-28 03:45:13.093 UTC [49] STATEMENT:  select check_password('juan',
'clave_1234');
```

Para más información sobre el manejo de errores ver:

- https://www.postgresql.org/docs/14/plpgsql-errors-and-messages.htm
- https://www.postgresql.org/docs/14/errcodes-appendix.html

Triggers

Básicamente es una indicación que podemos proporcionar al motor de base de datos para que ejecute una función ante determinado evento. La función que ejecute puede ser de cualquier tipo (sql, plpsql, plpythonu, etc) y debe haber sido creada previamente a la definición del trigger. La única salvedad es que la función no debe definir parámetros y su tipo de retorno debe ser trigger.

Tipos y Usos

PostgreSQL soporta triggers sobre tablas y vistas.

Sobre las tablas los tipos de triggers soportados son:

- BEFORE: se ejecuta antes que el evento para el cual se definió
- AFTER : se ejecuta inmediatamente después que el evento para el cual se definió

Los eventos sobre los cuales se pueden disparar son:

- INSERT
- UPDATE
- DELETE
- TRUNCATE

Los triggers pueden ser definidos para que ejecuten, la función que tienen definida, por cada fila o una sola vez por la instrucción invocada.

Algo a tener en cuenta es que en triggers sobre eventos de UPDATE se puede especificar que el trigger se dispare sí y sólo sí determinadas columnas son afectadas por dicha operación.

When	Event	Row-level	Statement-level
Before	insert/update/delete	Tables	Tables + Views
	truncate	-	Tables
After	insert/update/delete	Tables	Tables + Views
	truncate	-	Tables
Instead Of	insert/update/delete	Views	-
	truncate	-	-

Registros especiales

Como vimos, en la definición de la función que es ejecutada por un trigger, no debe recibir parámetros. Ahora bien, el motor de base de datos inyecta a estas funciones, de forma automática, algunos datos (registros) que dependen del evento.

Registro NEW

Este registro contiene la información de los datos a insertar o actualizar en la tabla en cuestión. Este registro es inyectado en la operaciones de insert o update sólo para triggers que se ejecutan a nivel de filas.

Este registro es NULL en operaciones de delete o para triggers que se ejecutan por operación.

Registro OLD

Este registro contiene la información de los datos eliminados en la tabla en cuestión. Este registro es inyectado en las operaciones de delete sólo para triggers que se ejecutan a nivel de filas.

Este registro es NULL en operaciones insert o update o para triggers que se ejecutan por operación.

Creación de triggers

La sintaxis básica para crear triggers es la siguiente:

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR
... ] }
ON table
[ FROM referenced_table_name ]
[ NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE | INITIALLY
DEFERRED } ]
[ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name } [ ... ] ]
[ FOR [ EACH ] { ROW | STATEMENT } ]
[ WHEN ( condition ) ]
EXECUTE PROCEDURE function_name ( arguments )
```

PostgreSQL admite especificar una condición booleana en la definición del trigger sobre la cual basa su decisión para ejecutar el trigger o no. Esta condición se especifica mediante la palabra ***WHEN***

Nota: si se definen varios triggers sobre la misma tabla y para el mismo evento, los mismos se ejecutarán en orden alfabético.

Nota II: puede existir recursividad en un trigger. Es responsabilidad del programador que no se creen loops infinitos.

Algunos ejemplos:

```
CREATE TRIGGER check_update
BEFORE UPDATE ON accounts
FOR EACH ROW
EXECUTE PROCEDURE check_account_update();
```

```
CREATE TRIGGER check_update
BEFORE UPDATE ON accounts
FOR EACH ROW
WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
EXECUTE PROCEDURE check_account_update();
```

```
CREATE OR REPLACE FUNCTION before_insert_provincias() RETURNS trigger AS $$

BEGIN
       RAISE INFO 'ejecuta trigger';
       IF length(new.prv_nombre) < 3 THEN
          return null;
       END IF;
       return new;
END;

$$ LANGUAGE plpgsql;


CREATE TRIGGER before_insert_provincias
BEFORE INSERT ON provincias
FOR EACH ROW
EXECUTE PROCEDURE befor_insert_provincias();
```

En triggers before insert o before update, cuando realizamos validaciones y las mismas no se cumplen, podemos devolver NULL. Esto hace que el registro que va a tratar de insertar o

actualizar posee todos sus campos en NULL, lo que generará un error al tratar de realizar la operación.

Lectura recomendada

- PostgreSQL Triggers. Disponible en: https://www.postgresqltutorial.com/postgresql-triggers

Bibliografía

- Manejo de errores. Disponible en: https://www.postgresql.org/docs/14/plpgsql-errors-and-messages.html
- Códigos de error. Disponible en: https://www.postgresql.org/docs/14/errcodes-appendix.html
- PostgreSQL Triggers Documentation. Disponible en: https://www.postgresql.org/docs/current/sql-createtrigger.html
- PostgreSQL Trigger Procedures. Disponible en: https://www.postgresql.org/docs/10/plpgsql-trigger.html#PLPGSQL-DML-TRIGGER

Diseño y Administración de Base de Datos

Unidad VI

Índices y Optimización de Consultas

Ing. Alejandro Mostovoi

CUVL - 2018

Índice

Índices **2**
- Qué es un índice 2
- Cómo funciona un índice 2
- Tipos de índices 3
- Clustering 5

Optimización de consultas **6**
- Scan Methods 7
- Join Methods 8
  - Nested loops 8
    - Sequential Scan 8
    - Index Scan 8
  - Hash Join 9
  - Merge Join 9

Bibliografía **11**

Índices

Qué es un índice

Definición	Un índice, es una estructura que almacena datos copiados del campo de la tabla sobre el cual se definió el índice.

Su función, al igual que el índice de un libro, es permitir encontrar algo de forma más eficiente, en un libro será un capítulo del libro mientras que en tabla será un registro de la misma.

La forma de almacenar los índices varía de DBMS a DBMS, lo importante es que todos establecen la separación en el nivel físico.

La sintaxis básica para crear un índice es:

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table [ USING method ]
( { column | ( expression ) }[ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
[WHERE predicate]
```

- **Concurrently**: permite crear el índice sin bloquear la tabla
- **Method**: uno de btree, hash, gin, gist, spgist, brin. (Si no se especifica asume btree)
- **Expression**: permite definir una expresión para crear el índice.
- **Predicate**: expresión para definir un índice parcial.

Cómo funciona un índice

Cuando realizamos una consulta a la base de datos, el *Procesador de Consultas* (el encargado de generar los planes de consultas) define si debe recorrer la tabla completa (escaneo secuencial) o si utilizará alguno de los índices definidos para obtener los registros solicitados de forma más rápida.

No siempre el acceso a través de los índices es el más rápido, por ejemplo si la tabla posee pocos registros siempre será más rápido un acceso secuencial que un acceso a través de alguno de los índices definidos.

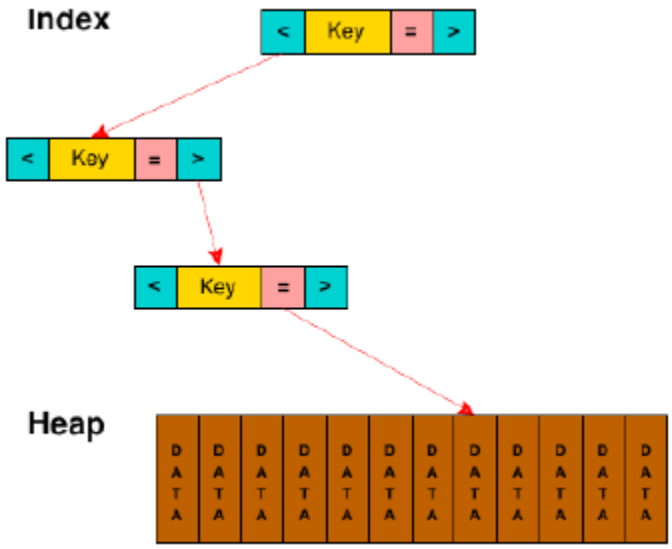
Otra cuestión a tener en cuenta es que cada actualización a una tabla (crear, modificar o eliminar un registro) impactará en los índices que la misma posea, por ende una tabla con muchos índices se traducirá en una baja en el rendimiento en las operaciones de alta, baja o modificación de registros.

Tipos de índices

Existen distintos tipos de índices, a continuación mencionamos algunos de los más comunes:

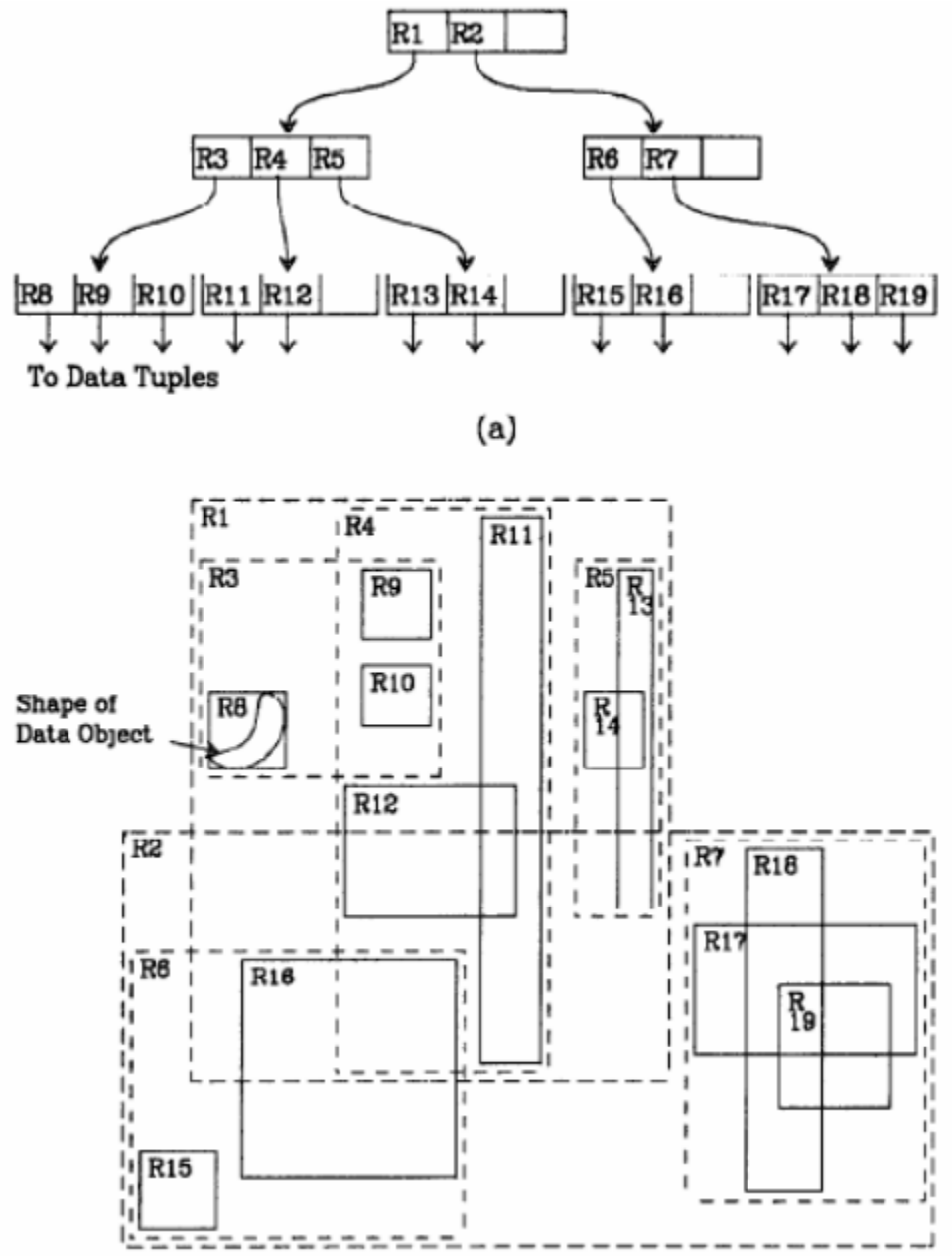
B-Tree

- Ideal sobre columnas que posean valores únicos.
- Puede proveer los datos ordenados
- Mejor performance en temas de concurrencia



Generalized Search Tree (GIST)

- Ideal para indexar datos complejos que no se “adaptan” a un índice b-tree
- Permite indexar datos geográficos, arrays, par clave/valor



Generalized Inverted Index (GIN)

- Ideal para indexar valores compuestos
- Ideal para indexar columnas con muchos duplicados, ya que por cada clave puede tener punteros a varias filas. Esto también reduce el tamaño del índice
- Este índice guarda un conjunto de pares (clave, valor) donde el “valor” es un conjunto de IDs de filas donde ocurre esa clave
- Muy utilizado en valores de tipo: documento, JSON, arrays

Otros

- Índices por expresiones (Expression Index)
    - Permite definir el índice a partir de una expresión, función, concatenación, etc.
    - Ej: lower(nombre)
- Índices parciales (Partial Index)
    - Permite definir un índice para un subconjunto de filas y no toda la tabla
    - Reduce el overhead en operaciones de insert/update
    - Reduce el tamaño del índice
    - Ej: CREATE INDEX state_idx ON customer (state) WHERE state = 'AZ';

Resumen

- Sólo B-tree, GIN y GIST soportan índices multicolumna
- B-tree ideal sobre columnas que posean valores únicos.
- GIN Ideal para indexar columnas con muchos duplicados
- GIST para todo lo demás

Clustering

Toda las estructuras que maneja una base de datos son almacenadas finalmente en disco, por lo tanto, mientras más rápido sea el acceso a disco, más rápido va a poder devolver los datos que se le piden. Una forma de posibilitar esto es guardar los datos de forma contínua o lo más "cercanos" entre sí que se pueda. A esto se lo llama clustering y las base de datos lo realizan a partir de un índice que debe ser creado previamente.

Cuando se están manejando datos espaciales, esto es realmente útil debido a lo costoso de procesar dichos datos.

Sintaxis

```
CLUSTER [VERBOSE] table_name [ USING index_name ]
```

Ejemplo

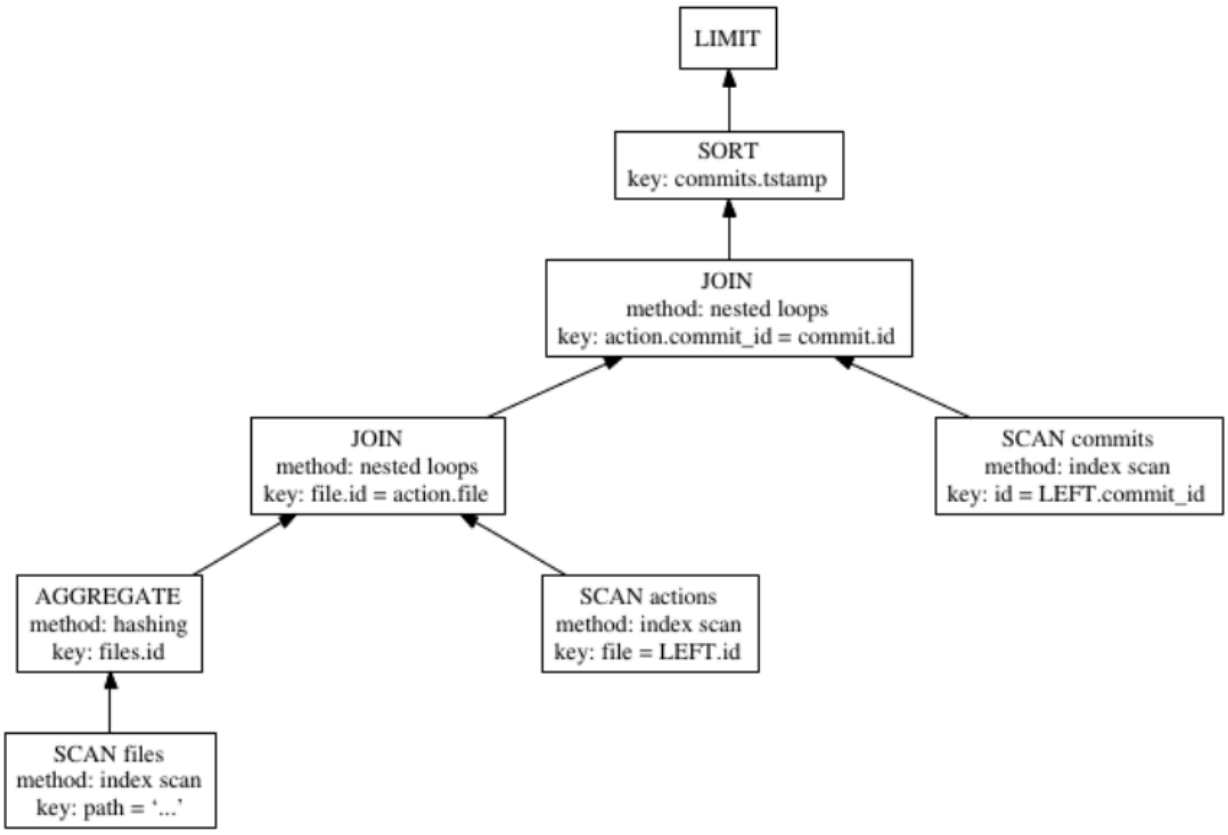
```
CREATE INDEX geom_idx ON TABLE my_tabla_geo USING GIST(the_geom);

CLUSTER my_tabla_geo USING geom_idx;
```

Optimización de consultas

SQL es un lenguaje declarativo, esto significa que podemos decirle QUÉ hacer, pero no CÓMO. Por lo tanto, el motor de BD por cada consulta evalúa distintos planes de ejecución seleccionando el menos costoso.

El comando EXPLAIN nos permite ver, evaluar y mejorar dichos planes.

Ejemplo



¿Cómo ejecutarlo?

```
EXPLAIN select * from table_dummy;

EXPLAIN VERBOSE select * from table_dummy;
```

¿Cómo interpretar la salida?

```
QUERY PLAN
-------------------------------------------------------------
Seq Scan on table_dummy (cost=0.00..458.00 rows=10000 width=244)
```

Para cada consulta, el motor de base de datos arma un árbol de ejecución, donde cada nodo es un plan a ejecutar.

Por cada plan de ejecución que el motor realice se mostrará una fila (un nodo del árbol).

Para interpretar la salida del comando EXPLAIN, estos nodos deben leerse desde abajo hacia arriba.

Cada nodo presenta la siguiente información:

- Costo estimado para comenzar a devolver información
- Costo estimado para finalizar la consulta
- Cantidad estimadas de filas procesadas
- Cantidad estimados de bytes

¿Qué tiene que evaluar el planner?

- Scan Methods (sequential, index, bitmap index)
- Join Methods (nested loops, hash join, merge join)
- Aggregation Methods (hash, sort)

Scan Methods

Son los métodos que utiliza el motor de base de datos para leer (obtener y devolver) los datos de las tablas. Puede ser secuencial o indexado.

¿Importa la distribución de los datos?

Sí. Al motor “le importa” porque en base a eso, también determina qué método de escaneo realizar sobre la tabla.

No siempre un acceso indexado es más eficiente que un acceso secuencial. Por ejemplo, si el resultado de la consulta involucra más del 70% de los datos de la misma, entonces un acceso secuencial es más eficiente que uno indexado.

¿Puedo forzar el uso de índices?

Sí. Con los siguientes comandos:

- SET enable_seqscan = false;
- SET enable_bitmapscan = false;
- SET enable_nestloop = false

Ante acciones como estas, hay que prestar atención al impacto sobre las consultas.

Corolario: en caso de que el motor siga eligiendo un plan que no involucra el uso de índices, puede deberse a causas más fundamentales, como por ejemplo que la condición del query no utilice campos del índice.

Join Methods

Son los métodos que utiliza el motor de base de datos para unir/cruzar los datos de las tablas. Es una de las cosas más importantes a tener en cuenta en la optimización de consultas.

Nested loops

Puede ser tanto *sequential scan* como *index scan*

Sequential Scan

Ocurre ante un join entre 2 tablas y lo que hace el motor es comparar todas las filas de la tabla 1 contra todas las filas de la tabla 2. Sólo es útil cuando las tablas involucradas tienen pocas filas ya que lo realiza de forma muy rápida, tratar de evitar en otros casos.

En pseudocódigo sería algo como:

```
for (i = 0; i < length(t1); i++)
  for (j = 0; j < length(t2); j++)
    if (t1[i] == t2[j])
      output(t1[i], t2[j]);
```

Index Scan

¿Qué ocurre cuando creamos índices sobre las tablas y realizamos algún join estableciendo un filtro? En este caso, el motor va a realizar un nested loop, pero utilizando la información proporcionada por los índices. Con lo cual sólo va recorrer las filas que verifiquen la condición establecida en ambas tablas.

```
for (i = 0; i < length(outer); i++)
  index_entry = get_first_match(outer[j])
while (index_entry)
  output(outer[i], inner[index_entry]);
  index_entry = get_next_match(index_entry);
```

Corolario	no todos los nested loops son malos. Si se tiene un índice sobre la tabla no es necesariamente malo

Hash Join

Lo que hace es crear una estructura en memoria donde guarda el hash de las claves de 1 tabla y luego recorre dicha estructura efectuando comparaciones contra los valores de la otra tabla involucrada. Es uno de los joins más comunes elegidos por el motor. Es bastante efectivo siempre y cuando la tabla sobre la cual se genera el hash pueda ser almacenada en memoria.

Por lo general ocurre cuando se establece algún tipo de restricción sobre alguna tabla. Restricciones del estilo > o < pero no =.

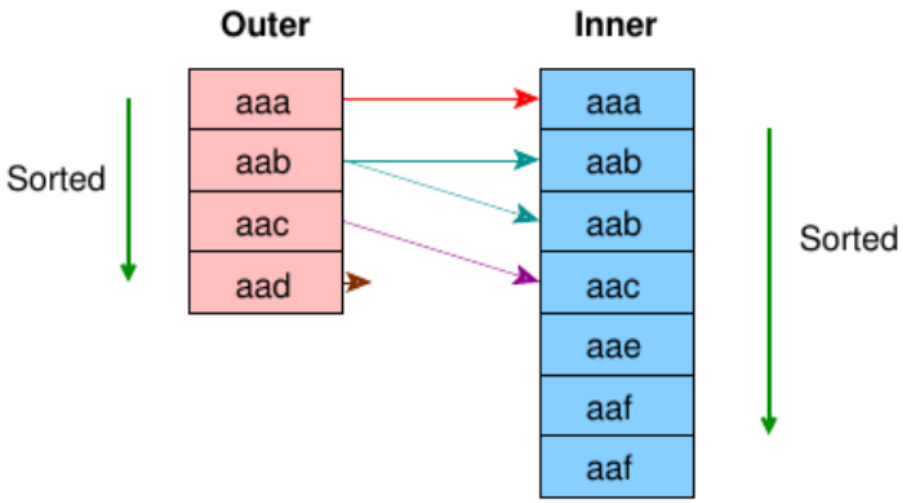
En pseudocódigo sería algo como:

```
for (j = 0; j < length(inner); j++)
  hash_key = hash(inner[j]);
  append(hash_store[hash_key], inner[j]);

for (i = 0; i < length(outer); i++)
  hash_key = hash(outer[i]);
  for (j = 0; j < length(hash_store[hash_key]); j++)
    if (outer[i] == hash_store[hash_key][j])
      output(outer[i], inner[j]);
```

Merge Join

Lo que hace el motor en este caso es ordenar ambas tablas y luego las recorre casi en paralelo devolviendo las filas que machean la condición del join. Este tipo de join ocurre cuando se establece un join sin una restricción. Es bastante útil para grandes tablas ya que no requiere que

la tabla sea levantada en memoria y porque no se realizan múltiples recorridas sobre las tablas. Tener en cuenta que si las tablas poseen algún tipo de índice que establece un orden, entonces el motor evita ordenar las tablas previamente.



Comandos que ayudan al motor a mejorar los planes de consultas:

- ANALYZE: ayuda a recolectar estadísticas sobre la distribución de los datos
- VACUUM: limpia índices y tablas.

¿Cuál es la importancia de estos comandos?

Ayudan al motor a tener estadísticas sobre las tablas y sus datos y por ende a mejorar el plan de acción de cada consulta.

Sitio recomendado

- https://explain.depesz.com/

Bibliografía

- PostgreSQL Current Documentation. Disponible en: https://www.postgresql.org/docs
- Explaining the Postgres Query Optimizer. Disponible en http://momjian.us/main/writings/pgsql/optimizer.pdf
- Explaining the Postgres Query Optimizer Disponible en https://www.youtube.com/watch?v=P5iZri9s0WQ